

THINK C™

The Professional's Choice

Standard
Libraries
Reference

THINK C

The Professional's Choice

STANDARD
LIBRARIES
REFERENCE

Credits

Standard Libraries Reference: Jeff Mattson
Jon Burrowes

Software: Michael Kahl and Jörg 'jbx' Brown

Product Manager: Diana Bury

THINK Class Library: Gregory H. Dow

Copyright © 1989 Symantec Corporation. All Rights Reserved.

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
408/253-9600

Technical Support: 617/275-1710

The product names mentioned in this manual are the trademarks or registered trademarks of their manufacturers.

ResEdit and RMaker are copyrighted programs of Apple Computer, Inc. licensed to Symantec Corp. to distribute for use only in combination with THINK C. Apple software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of execution of THINK C. When THINK C has completed execution, Apple Software shall not be used by any other program.

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED REGARDING THE ENCLOSED SOFTWARE PACKAGE, IT'S MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED IN SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE THAT VARY FROM STATE TO STATE.

Contents

1	The Libraries	1
2	ANSI Header Files	3
	What's in the Listing.....	3
	Listing of Header Files	4
3	ANSI.....	23
	What Is the ANSI Library?.....	23
	The Console Package.....	23
	Other ANSI Library Configurations.....	24
	Catalogue of Functions	25
4	console.....	165
	What Is the Console Library?.....	165
	What Is a Console?	165
	Environments.....	166
	Options	167
	Catalogue of Functions	169
5	unix.....	183
	What Is the Unix Library?.....	183
	Catalogue of Functions	185
	Appendices	
A	printf() / scanf().....	203
	Structure of Format Arguments.....	203
	Format Arguments for printf().....	204
	Format Arguments for scanf().....	209
	Common Errors	212
	Index	213
	License Agreement.....	219

The Libraries

1

Introduction

This manual describes these libraries included with THINK C:

- ANSI
- console
- unix

Topics covered in this chapter:

- The libraries

The Libraries

The libraries are supplied in project form so the smart linker can create the smallest code possible. The source code for all the standard libraries is in your THINK C package, so you can modify the routines to suit your programs.

This is the complete list of libraries described in this manual (found in the C Libraries folder):

ANSI	The ANSI standard library.
ANSI-881	The ANSI standard library for machines with the MC68881.
ANSI-small	A subset of ANSI, without console or floating-point support.
ANSI-A4	A subset of ANSI-small, for non-applications (drivers, etc.).
console	Advanced console-handling functions.
unix	Common Unix routines not in the ANSI standard library.

A function index is in the back of this manual to help you quickly locate any function you need.

ANSI Header Files

2

Introduction

This chapter lists the contents of the header files for the ANSI library.

Topics covered in this chapter:

- What's in the listing
- A listing of the ANSI header files

What's in the Listing

The following listing contains the various types, macros, and functions declared in the header files for the ANSI library.

For each header file...

For most macros describing constant values, the listing includes their values in THINK C and descriptions. Floating-point values are rounded to five decimal places.

For most macros with arguments, the listing includes just a description. Most have more extensive documentation in Chapter 3, "ANSI."

For most types, the listing includes their definitions in THINK C and descriptions. Type definitions for the lengthier structs (generally, those that don't fit in one line) aren't included.

For functions, the listing includes just the name. More extensive documentation is in Chapter 3, "ANSI."

The header files

These header files are for the ANSI library:

<code>assert.h</code>	<code>signal.h</code>
<code>ctype.h</code>	<code>stdarg.h</code>
<code>errno.h</code>	<code>stddef.h</code>
<code>float.h</code>	<code>stdio.h</code>
<code>limits.h</code>	<code>stdlib.h</code>
<code>locale.h</code>	<code>string.h</code>
<code>math.h</code>	<code>time.h</code>
<code>setjmp.h</code>	

<assert.h>

DESCRIPTION	For putting diagnostic messages in your programs.
TYPES	None.
MACROS	<code>assert()</code> For more information, read the <code>assert()</code> section in the ANSI chapter
FUNCTIONS	None.

<ctype.h>

DESCRIPTION

For testing and mapping characters.

TYPES

None.

MACROS

None.

FUNCTIONS

These functions are declared in `ctype.h`:

<code>isalnum()</code>	<code>isalpha()</code>	<code>isctrl()</code>
<code>isdigit()</code>	<code>isgraph()</code>	<code>islower()</code>
<code>isprint()</code>	<code>ispunct()</code>	<code>isspace()</code>
<code>isupper()</code>	<code>isxdigit()</code>	<code>tolower()</code>
<code>toupper()</code>		

<errno.h>

DESCRIPTION

Macros for error-reporting.

TYPES

None.

MACROS

Macro	Value	Description
EDOM	33	A domain error occurred, i.e., the value passed to this function was erroneous.
ERANGE	34	A range error occurred, i.e., the value that this function generated was erroneous.
errno	Contains an error number, which various functions set when they encounter errors.	

FUNCTIONS:

None.

<float.h>**DESCRIPTION**

Limits and other parameters for floating-point numbers.

TYPES

None.

MACROS

Macro	Value	Description
FLT_RADIX	2	The radix for an exponent in a floating-point number
FLT_ROUNDS	1	A code specifying how floating-point functions round the results of addition. THINK C rounds to the nearest integer.

These macros define some limits and other parameters for floating-point numbers. Macros beginning with FLT are for numbers of type float; DBL, double; and LDBL, long double. In this chart, all floating-point values are rounded to five decimal places.

Macro	Value	Description
FLT_MANT_DIG	24	The number of base FLT_RADIX digits in the floating-point significand.
DBL_MANT_DIG	64	
LDBL_MANT_DIG	64	
FLT_DIG	6	Decimal digits of precision
DBL_DIG	18	
LDBL_DIG	18	
FLT_MIN_EXP	-125	The largest integer that can be a negative exponent, when the radix is FLT_RADIX.
DBL_MIN_EXP	-16382	
LDBL_MIN_EXP	-16382	
FLT_MIN_10_EXP	-37	The largest integer that can be a negative exponent, when the radix is 10.
DBL_MIN_10_EXP	-4931	
LDBL_MIN_10_EXP	-4931	
FLT_MAX_EXP	128	The largest integer that can be a positive exponent, when the radix is FLT_RADIX.
DBL_MAX_EXP	16384	
LDBL_MAX_EXP	16384	

Standard Libraries Reference

FLT_MAX_10_EXP	38	The largest integer that can be a positive exponent, when the radix is 10.
DBL_MAX_10_EXP	4932	
LDBL_MAX_10_EXP	4932	
FLT_MAX	3.40282E+38	The largest representable floating-point number
DBL_MAX	1.18973E+4932	
LDBL_MAX	1.18973E+4932	
FLT_EPSILON	1.19209E-7	The smallest fraction that can be represented in floating point notation
DBL_EPSILON	1.08420E-19	
LDBL_EPSILON	1.08420E-19	
FLT_MIN	1.17549E-38	The smallest normalized positive floating-point number.
DBL_MIN	1E-4915	
LDBL_MIN	1E-4915	

FUNCTIONS:

None.

<limits.h>**DESCRIPTION**

Limits and other parameters for integers.

TYPES

None.

MACROS

These macros define limits and other parameters for integers:

Macro	Value	Description
CHAR_BIT	8	Number of bits for the smallest object that is not a bit-field.
MB_LEN_MAX	1	The maximum number of characters that can be in a multi-byte character.
SCHAR_MIN	-128	Minimum value for a signed char
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for a unsigned char
CHAR_MIN	-128	Minimum value for a char
CHAR_MAX	127	Maximum value for a char
SHRT_MIN	-32768	Minimum value for a short int
SHRT_MAX	32767	Maximum value for a short int
USHRT_MAX	65535	Maximum value for an unsigned short int
INT_MIN	-32768	Minimum value for an int
INT_MAX	32767	Maximum value for an int
UINT_MAX	65535	Maximum value for an unsigned int
LONG_MIN	-2147483648	Minimum value for a long int
LONG_MAX	2147483647	Maximum value for a long int
ULONG_MAX	4294967295	Maximum value for an unsigned long int

FUNCTIONS

None.

<locale.h>

DESCRIPTION

For localizing programs.

Note: This is provided for ANSI C compatibility only. THINK C supports only the minimal "C" locale.

TYPES

`lconv` A structure containing information related to the formatting of numeric values.

MACROS

These macros are for use in the `category` argument of the `setlocale()` function:

Macro	Value	Description
<code>LC_ALL</code>	-1	When set, causes all the categories to comply with the rules for the same locale.
<code>LC_COLLATE</code>	0	Affects the behavior of the <code>strcoll()</code> and <code>strxfrm()</code> functions.
<code>LC_TYPE</code>	1	Affects the behavior of the character-handling functions and the multi-byte functions.
<code>LC_MONETARY</code>	2	Affects the monetary formatting information that <code>localeconv()</code> returns.
<code>LC_NUMERIC</code>	3	Affects the decimal point character for the formatted output and string conversion functions.
<code>LC_TIME</code>	4	Affects the behavior of the <code>strftime()</code> function.

FUNCTIONS:

These functions are declared in `locale.h`:

`localeconv()` `setlocale()`

<math.h>**DESCRIPTION**

Declares several mathematical functions.

TYPES

None.

MACROS

Macro	Value	Description
HUGE_VAL	INF	Represents Infinity.

FUNCTIONS

These functions are declared in `math.h`:

<code>acos()</code>	<code>asin()</code>	<code>atan()</code>
<code>atan2()</code>	<code>ceil()</code>	<code>cos()</code>
<code>cosh()</code>	<code>exp()</code>	<code>fabs()</code>
<code>floor()</code>	<code>fmod()</code>	<code>frexp()</code>
<code>ldexp()</code>	<code>log()</code>	<code>log10()</code>
<code>modf()</code>	<code>pow()</code>	<code>sin()</code>
<code>sinh()</code>	<code>sqrt()</code>	<code>tan()</code>
<code>tanh()</code>		

<setjmp.h>

DESCRIPTION

For non-local jumps.

TYPES

Type	Definition	Description
jmp_buf	long X[11]	An array suitable for holding the information that is necessary to restore a calling environment.

MACROS

None.

FUNCTIONS

These functions are declared in set jmp . h:

longjmp () set jmp ()

<signal.h>**DESCRIPTION**

For handling signals.

TYPES

Type	Definition	Description
sig_atomic_t	int	The largest type that can be written without being interrupted, e.g., by a signal.

MACROS

Use these macros as the second argument to (and with the return value from) `signal()`. These macros compare unequally to the address of any declarable function:

Macro	Value	Description
SIG_DFL	0	Perform default handling of the signal.
SIG_ERR	-1	<code>signal()</code> reports this as its error when it fails.
SIG_IGN	1	Ignore the signal.

Use these macros as the first argument (which is the signal) to the `signal()` call. They expand to the signal numbers that correspond to the conditions listed here:

Macro	Value	Description
SIGABRT	1	Abnormal termination
SIGFPE	2	An erroneous arithmetic operation, e.g. a zero divide.
SIGILL	3	An invalid function image, e.g., an illegal instruction.
SIGINT	4	Interactive attention, e.g., interrupt.
SIGSEGV	5	Invalid access to storage.
SIGTERM	6	The sender wants the receiver to terminate itself.

FUNCTIONS

These functions are declared in `signal.h`:

`raise()` `signal()`

<stdarg.h>

TYPES

Type	Definition	Description
<code>va_list</code>	<code>void *</code>	Variable argument list. Holds a list of arguments that may vary in number and type.

MACROS

These macros are declared in `stdarg.h`:

<code>va_start()</code>	<code>va_arg()</code>	<code>va_end()</code>
-------------------------	-----------------------	-----------------------

For more information on these macros, read their section in the ANSI chapter.

FUNCTIONS

None.

<stddef.h>

DESCRIPTION

Some commonly-used definitions.

TYPES

Type	Definition	Description
ptrdiff_t	long	The result of subtracting two pointers.
size_t	unsigned long	The size of a C object.
wchar_t	char	A member of an extended character set.

Note: In THINK C, the type that sizeof returns is int, not size_t.

MACROS

Macro	Value	Description
NULL	(void *) 0	The NULL pointer constant.
offsetof()	The expression has the value of the offset from the beginning of a structure to the beginning of a structure member. The units of the offset are in bytes. This takes two arguments: the structure type <i>type</i> and the member name <i>member-designator</i> .	

FUNCTIONS

None.

<stdio.h>

DESCRIPTION

For performing input and output.

TYPES

Type	Definition	Description
size_t	unsigned long	The size of a C object (Original declaration is in stddef.h.)
fpos_t	unsigned long	An object that can uniquely specify every position within a file.
FILE	A structure to record all the information necessary to control a stream.	

MACROS

These macros are defined in stdio.h:

Macro	Value	Description
NULL	(void *) 0	The NULL pointer. (Original declaration is in stddef.h.)
BUFSIZ	512	The size of the buffer that setbuf() uses.
EOF	-1	The end-of-file.
FOPEN_MAX	15	The maximum number of files that can be open simultaneously.
FILENAME_MAX	256	The length of the longest filename, in characters.
L_tmpnam	20	The length of the longest filename string tmpnam() can generate.
TMP_MAX	9999	The maximum number of unique filenames that tmpnam() can generate.

These are the standard I/O streams:

<code>stderr</code>	The standard error stream.
<code>stdin</code>	The standard input stream.
<code>stdout</code>	The standard output stream.

Use these in `fseek()` calls as the third argument, specifying the place from which to start seeking:

Macro	Value	Description
<code>SEEK_SET</code>	0	The beginning of the file
<code>SEEK_CUR</code>	1	The current value of the file position indicator
<code>SEEK_END</code>	2	The end of the file

Use these in `setvbuf()` calls as the third argument, indicating how to buffer:

Macro	Value	Description
<code>_IOFBF</code>	0	Fully buffered.
<code>_IOLBF</code>	1	Line buffered
<code>_IONBF</code>	2	Not buffered.

FUNCTIONS:

These functions are declared in `stdio.h`:

<code>__getc()</code>	<code>__putc()</code>	<code>clearerr()</code>
<code>fclose()</code>	<code>feof()</code>	<code>ferror()</code>
<code>fflush()</code>	<code>fgetc()</code>	<code>fgetpos()</code>
<code>fgets()</code>	<code>fopen()</code>	<code>fprintf()</code>
<code>fputc()</code>	<code>fputs()</code>	<code>fread()</code>
<code>freopen()</code>	<code>fscanf()</code>	<code>fseek()</code>
<code>fsetpos()</code>	<code>ftell()</code>	<code>fwrite()</code>
<code>getc()</code>	<code>getchar()</code>	<code>gets()</code>
<code>perror()</code>	<code>printf()</code>	<code>putc()</code>
<code>putchar()</code>	<code>puts()</code>	<code>remove()</code>
<code>rename()</code>	<code>rewind()</code>	<code>scanf()</code>
<code>setbuf()</code>	<code>setvbuf()</code>	<code>sprintf()</code>

Standard Libraries Reference

<code>sscanf()</code>	<code>tmpfile()</code>	<code>tmpnam()</code>
<code>ungetc()</code>	<code>vfprintf()</code>	<code>vprintf()</code>
<code>vsprintf()</code>		

These functions are also defined as macros:

<code>feof()</code>	<code>ferror()</code>	<code>getc()</code>
<code>getchar()</code>	<code>putc()</code>	<code>putchar()</code>

<stdlib.h>**DESCRIPTION**

General utilities

TYPES

Type	Definition	Description
div_t	struct {int quot, rem}	What div() returns.
ldiv_t	struct {long quot, rem}	What ldiv() returns.
size_t	unsigned long	The size of a C object (Original declaration is in stddef.h.)
wchar_t	char	A member of an extended character set. (Original dec- laration in stddef.h.)

MACROS

Macro	Value	Description
EXIT_FAILURE	-1	Return unsuccessfully. Use as an argument to exit().
EXIT_SUCCESS	0	Return successfully. Use as an argument to exit().
RAND_MAX	3267	The maximum value that rand() will return.
MB_CUR_MAX	1	The maximum number of bytes in a multi-byte character.

FUNCTIONS

These functions are declared in stdlib.h:

abort()	abs()	atexit()
atof()	atoi()	atol()
bsearch()	calloc()	div()
exit()	free()	getenv()
labs()	ldiv()	malloc()
mblen()	mbstowcs()	mbtowc()
qsort()	rand()	realloc()
srand()	strtod()	strtol()
strtoul()	system()	wcstombs()
wctomb()		

<string.h>

DESCRIPTION

For handling strings.

TYPES

Type	Definition	Description
size_t	long int	The size of a C object. (Original declaration is in stddef.h.)

MACROS

Macro	Value	Description
NULL	(void *) 0	The NULL pointer. (Original declaration is in stddef.h.)

FUNCTIONS

These functions are declared in string.h:

memchr()	memcmp()	memcpy()
memmove()	memset()	strcat()
strcfm()	strchr()	strcmp()
strcoll()	strcpy()	strcspn()
strerror()	strlen()	strncat()
strncmp()	strncpy()	strpbrk()
strrchr()	strspn()	strstr()
strtok()		

<time.h>**DESCRIPTION**

For manipulating time.

TYPES

Type	Definition	Description
clock_t	unsigned long	The clock time.
time_t	unsigned long	The calendar time.
tm	A structure for a broken-down time, which includes the month, day, year, hour, minute, and second.	

MACROS

Macro	Value	Description
NULL	(void *) 0	The NULL pointer. (Original declaration is in stddef.h.)

FUNCTIONS

These functions are declared in time.h:

asctime()	clock()	ctime()
difftime()	gmtime()	localtime()
mktime()	strftime()	time()

Standard Libraries Reference

ANSI

3

Introduction

This chapter describes THINK C's ANSI library and includes a complete catalogue of the functions it contains.

Topics covered in this chapter:

- What is the ANSI library?
- The console package.
- Other ANSI library configurations.
- The catalogue of functions.

What Is the ANSI Library?

THINK C's ANSI library is a complete implementation of the ANSI standard library. It includes functions to perform file and screen I/O, string-handling, math, and more. The standard gives all C programmers in all ANSI C environments a consistent set of functions, making programs easier to port.

Using the ANSI library

To use a function in the ANSI library, add the ANSI library to your project. Then, look up the function in this chapter's catalogue, note the header file #included in the Syntax section, and #include that file in your file.

Multi-byte characters and locales

For the multi-byte character and locale functions, THINK C implements only the minimum functionality required by the ANSI standard. All the multi-byte character and locale functions are in the ANSI library. But the maximum length for a multi-byte character is one byte. Similarly, the only locale THINK C supports is the minimal "C" locale.

The Console Package

THINK C's ANSI library includes the console package. This package emulates a glass terminal and lets you easily port programs from a Unix or MS-DOS environment to the Macintosh.

If your application performs I/O to the `stdin`, `stdout`, or `stderr` streams, the console package creates a window and menu bar for you. The I/O the application performs to these

streams (using, for example, `printf()` or `getc()`) happens in the console window. The menu bar gives you access to all your desk accessories and the ability to cut to and paste from the console window. For an example of an application that uses the console package in the ANSI library, see Chapter 3 of the *THINK C User's Manual*, "Tutorial: Hello World."

Using the console library, you can have more control over the console window and create even more of them. For more information, see Chapter 4, "console."

You can use the ANSI library in applications that don't use the console package. For example, you may want to use the string-handling functions (such as `sprintf()` or `strcmp()`) or math functions (such as `atan()`) in the ANSI standard library. As long as you do not perform I/O to the `stdin`, `stdout`, or `stderr` streams, the console package will not be invoked.

Other ANSI Library Configurations

These other libraries are available for certain special cases.

ANSI-881

The ANSI-881 library is for applications that will run only on Macintoshes with the MC68881 math co-processor. All the floating-point functions in this library use the MC68881.

ANSI-small

The ANSI-small library is for applications that do not use the full functionality of the complete ANSI library. It is smaller than the ANSI library, and applications that use ANSI-small will probably be smaller as well.

This library does not contain the console package and the floating-point functions. Also, the `%f` floating-point conversion is not available for `printf()`, `scanf()`, and other related functions.

ANSI-A4

The ANSI-A4 library is a subset of the ANSI-small library for non-applications (desk accessories, code resources, etc.). The ANSI, ANSI-881, and ANSI-small libraries should be used in applications only.

Not only does this library exclude the console package, the floating-point functions, and the `%f` conversion; it also excludes all functions that cannot be used outside of applications, such as file I/O functions. This library does include the string handling functions, such as `sprintf()` and `sscanf()`, in addition to many other functions.

_atexit

LIBRARY

ANSI

SYNTAX

```
#include<stdlib.h>
int _atexit (void (*func) (void));
```

DESCRIPTION

`_atexit()` adds the function pointed by `func` to a list of functions to be called without arguments at normal or abnormal program termination.

A program terminates abnormally if it calls `abort()`, `_exit()`, or `ExitToShell()`

This function is a THINK C extension that is not in the ANSI standard.

RETURNS

Zero, if the routine could add the function to the list.

Non-zero, if it could not.

SEE ALSO

`atexit()`

`_exit`

LIBRARY

ANSI

SYNTAX

```
#include<stdlib.h>
void _exit (int status);
```

DESCRIPTION

`_exit ()` terminates the program abnormally.

`_exit ()` executes all functions registered with `_atexit ()`, flushes all open output streams, closes all open files, and removes all temporary files.

The `status` argument is ignored.

This function is a THINK C extension that is not in the ANSI standard.

SEE ALSO

`exit ()`, `abort ()`

`_exiting`

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void _exiting(int i);
```

DESCRIPTION

`_exiting()` executes the routines done when a program exits without exiting a program. You can use `_exiting()` when you use `Launch()` or any other routine that exits a program without executing the exiting routines.

Whether `_exiting()` performs the routines for a normal or an abnormal exit is determined by the value of `i`:

If <code>i</code> is...	<code>_exiting()</code> executes the routines for a...
1	Normal exit (i.e., executes all functions registered with <code>_atexit()</code> or <code>atexit()</code> , flushes all open output streams, closes all open files, and removes all temporary files.)
0	Abnormal exit (i.e., executes all functions registered with <code>_atexit()</code> , flushes all open output streams, closes all open files, and removes all temporary files.)

This function is a THINK C extension that is not in the ANSI standard.

SEE ALSO

`_exit()`, `abort()`, `exiting()`

abort

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void abort(void);
```

DESCRIPTION

`abort()` causes the calling program to terminate abnormally.

`abort()` executes all functions registered with `_atexit()`, flushes all open output streams, closes all open files, and removes all temporary files.

`abort()` returns no special information to the host environment to denote unsuccessful/abnormal program termination.

If `abort()` cannot abort the program, it sends the program the signal `SIGABRT` (whose purpose is essentially notification).

If the program responds to `SIGABRT`, and if the resulting signal handler function does not return (e.g., calls `longjmp()`), the program will not terminate.

abs

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
int abs(int i);
```

DESCRIPTION

abs () computes the absolute value of its argument.

RETURNS

| i | for any integer i.

acos

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double acos(double x);
```

DESCRIPTION

`acos()` computes the principal value of the arc cosine ($\cos^{-1}$) of x , for any x in the range -1 to $+1$.

RETURNS

$\cos^{-1}x$ in the range 0 to π radians, if successful

0 (zero), if $|x|$ is greater than 1 . It also sets `errno` to `EDOM`.

asctime

LIBRARY

ANSI

SYNTAX

```
#include<time.h>
char *asctime(struct tm *timeptr)
```

DESCRIPTION

asctime() converts into a string the broken-down time that resides in the structure that `timeptr` points to. The string has the form (for example):

```
Tue Mar 11 01:03:52 1989\n\0
```

RETURNS

A pointer to the string containing the time.

asin

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double asin(double x);
```

DESCRIPTION

`asin()` computes the principal value of the arc sine ($\sin^{-1}$) of x , for any x in the range -1 to +1.

RETURNS

$\sin^{-1}x$ in the range $-\pi/2$ to $+\pi/2$ radians, if successful.

0 (zero), if $|x|$ is greater than 1. It also sets `errno` to `EDOM`.

assert

LIBRARY

ANSI

SYNTAX

```
#include <assert.h>
void assert(int expression);
```

DESCRIPTION

`assert()` asserts an expression. If the expression is false (that is, equals 0 (zero)), `assert()` prints an error message to `stderr` about this failed call and calls `abort()`.

`assert()` is useful for putting diagnostics into a program. It reports its failure in the following format:

Assert failed: *expression*, file *filename*, line *num*

To remove the `assert()` calls when you're finished debugging, you don't need to remove the actual statements. Instead, use one of these methods:

- To remove them from a whole file, place a `#define NDEBUG` statement just before the `#include <assert.h>` statement.
- To remove them from a region of a file, place a `#define NDEBUG` statement at the beginning of the region, and a `#undef NDEBUG` statement at the end of the region. Be sure that the region does not contain the `#include <assert.h>` statement.

atan

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double atan(double x);
```

DESCRIPTION

`atan()` computes the principal value of the arc tangent ($\tan^{-1}$) of x .

RETURNS

$\tan^{-1}x$, in the range $-\pi/2$ to $+\pi/2$ radians.

atan2

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double atan2(double y, double x);
```

DESCRIPTION

`atan2()` computes the principal value of the arc tangent ($\tan^{-1}$) of y/x . It uses the signs of both arguments to determine the quadrant of the return value.

RETURNS

$\tan^{-1}(y/x)$ in the range $-\pi$ to $+\pi$ radians, if successful.

0 (zero), if both x and y are 0 (zero). It also sets `errno` to `EDOM`.

atexit

LIBRARY

ANSI

SYNTAX

```
#include<stdlib.h>
int atexit (void (*func) (void));
```

DESCRIPTION

`atexit()` adds the function pointed by `func` to a list of functions to be called without arguments at normal program termination. This function will *not* be called at abnormal program termination; i.e., when the program calls `abort()`, `_exit()`, or `ExitToShell()`.

These are the conditions for normal program termination:

- The program returns from `main()`
- The program calls `exit()`.

SEE ALSO

`_atexit()`

atof

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
double atof(char *nptr)
```

DESCRIPTION

atof() converts the string to which nptr points to its double-precision floating point equivalent.

RETURNS

The converted double-precision floating point value.

atof

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
double atof(char *nptr)
```

DESCRIPTION

`atof()` converts the string to which `nptr` points to its double-precision floating point equivalent.

RETURNS

The converted double-precision floating point value.

atoi

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
int atoi(char *nptr)
```

DESCRIPTION

atoi () converts the string to which nptr to its integer equivalent.

RETURNS

The converted integer value.

atol

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
long int atol(char *nptr)
```

DESCRIPTION

atol() converts the string to which nptr points to its long integer equivalent.

RETURNS

The converted long integer value.

bsearch

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void *bsearch(void *key,
              void *base,
              size_t nmemb,
              size_t size,
              int (*compar) (void *, void *));
```

DESCRIPTION

This performs a binary search of a sorted table.

`bsearch()` calls the comparison function that `compar` points to with two arguments. The arguments point to the key object and to an array element, in that order.

The comparison function returns an integer as follows:

- less than zero, if the key object is less than the array element.
- zero, if the key object is equal to the array element.
- greater than zero, if the key object is greater than the array element.

It's good practice to make sure the array is sorted before passing it to `bsearch()`.

These are the arguments to `bsearch()`:

<code>key</code>	Points to the object that <code>bsearch</code> will try to match.
<code>base</code>	Initial element of the array.
<code>nmemb</code>	Number of objects in the array.
<code>size</code>	Size of each element in the array.
<code>compar()</code>	Pointer to the comparison function.

RETURNS

A pointer to the element of the array that matches the key, if it finds a match.

NULL, if it doesn't find a match.

calloc

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

DESCRIPTION

`calloc()` allocates space for an array of `nmemb` objects, each of whose size is `size`. It initializes all bits in the space to zero.

Use `free()` to deallocate a block of memory that was allocated by `calloc()`.

RETURNS

A pointer to the allocated space, if successful.

NULL, if failure.

ceil

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double ceil(double x);
```

DESCRIPTION

`ceil()` rounds `x` up to the next highest integer. For example, `ceil(1.1)` is 2, and `ceil(4.9)` is 5.

RETURNS

A double that is the next highest integer from `x`.

clearerr

LIBRARY

ANSI

SYNTAX

```
#include <stdio>
void clearerr(FILE *stream);
```

DESCRIPTION

clearerr() clears the end-of-file and error indicators for the stream that stream points to.

SEE ALSO

ferror(), feof(), rewind()

clock

LIBRARY

ANSI

SYNTAX

```
#include <time.h>
clock_t clock(void)
```

DESCRIPTION

`clock()` determines the elapsed time since power-up, in clock ticks.

The time in seconds is `clock() / CLOCKS_PER_SEC`.

`clock()` is useful for calculating durations.

RETURNS

The elapsed time since power-up, in clock ticks, if successful

`(clock_t) -1`, if that time is not available or its value cannot be represented.

COS**LIBRARY**

ANSI

SYNTAX

```
#include <math.h>
double cos(double x);
```

DESCRIPTION

`cos()` computes the cosine of the argument `x`.

RETURNS

`cos x`, in radians.

cosh

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double cosh(double x);
```

DESCRIPTION

`cosh()` computes the hyperbolic cosine of the argument `x`.

RETURNS

`cosh x`, in radians, if successful,

Infinity, if `x` is too large. It also sets `errno` to `ERANGE`.

ctime

LIBRARY

ANSI

SYNTAX

```
#include <time.h>
char *ctime(time_t *timer);
```

DESCRIPTION

`ctime()` converts to local time the calendar time `timer` points to. The result is a 26-character string in the form (for example):

```
Sun Sep 16 01:03:52 1973\n\0
```

`ctime()` is equivalent to `asctime(localtime(timer))`.

If `timer` is `NULL`, `ctime()` takes the time from the Macintosh's time buffer.

RETURNS

A pointer to the string containing the local time.

difftime

LIBRARY

ANSI

SYNTAX

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

DESCRIPTION

`difftime()` computes the difference between two calendar times, `time1-time0`.

RETURNS

The difference between two calendar times, in seconds, as a double.

div

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
div_t div(int numer, int denom)
```

DESCRIPTION

`div()` computes the quotient and remainder of the division of the numerator `numer` by the denominator `denom`; that is:

`quot * denom + rem == numer.`

If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient.

This returns a function of type `div_t`, which looks like this:

```
struct {
    int quot; /* quotient */
    int rem;  /* remainder */
}
```

RETURNS

A structure of type `div_t` that contains the results of the division.

exit

LIBRARY

ANSI

SYNTAX

```
#include<stdlib.h>
void exit (int status);
```

DESCRIPTION

`exit ()` terminates the program normally.

`exit ()` executes all functions registered with `_atexit ()` or `atexit ()`, flushes all open output streams, closes all open files, and removes all temporary files.

The `status` argument is ignored.

SEE ALSO

`_exit ()`, `abort ()`

exp

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double exp(double x);
```

DESCRIPTION

`exp()` computes e (the base of natural logarithms) to the x th power.

RETURNS

e^x , if successful

Infinity, if the argument is too large for a double-precision floating point number. It also sets `errno` to `ERANGE`.

fabs

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double fabs(double x);
```

DESCRIPTION

`fabs()` computes the absolute value of a floating-point number, `x`.

RETURNS

`| x |`, for any value `x`.

fclose

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fclose(FILE *stream);
```

DESCRIPTION

`fclose()` flushes the stream that `stream` points to and closes the file that is associated with that stream. `fclose()` delivers any unwritten buffered data to the host environment and discards any unread buffered data.

`fclose()` won't close or deallocate buffers that the program allocated itself (e.g., by handing a buffer to `setbuf()`) but it will close buffers that were allocated automatically (e.g., by `setbuf()`, if the program called `setbuf()` but did not hand it a buffer, or by `fopen()`).

The difference between `fclose()` and the Unix Library function `close()` is that `fclose()` takes a file pointer as its argument, while `close()` takes a file descriptor number.

RETURNS

0 (zero), if successful.

EOF, if failure.

feof

LIBRARY

ANSI

SYNTAX

```
#include <stdio>
int feof(FILE *stream);
```

DESCRIPTION

`feof()` tests the end-of-file indicator for the stream that `stream` points to.

`feof()` is useful to test when a previous I/O call to the given stream caused an EOF error (such as trying to get a character after having reached the end of the file).

Note: If you use `clearerr()` to clear the error flag of a stream, `feof()` will no longer report an EOF error. Similarly, a later call with another type of error will also erase the EOF error flag. (`fseek()` clears the EOF indicator.)

RETURNS

Non-zero, if at the end of the file.

Zero, otherwise.

ferror

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int ferror(FILE *stream);
```

DESCRIPTION

`ferror()` tests the error indicator for the stream that `stream` points to.

To clear the error, use `clearerror()`. (`rewind()` clears the error indicator for a stream automatically.)

RETURNS

Non-zero, if the stream's error indicator is set.

Zero, otherwise.

SEE ALSO

`feof()`, `clearerr()`, `rewind()`

fflush

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fflush(FILE *stream)
```

DESCRIPTION

`fflush()` flushes the I/O buffer of the given stream. `fflush()` allows the program to switch cleanly between reading and writing on the same stream.

For streams on which the program was just writing, `fflush()` causes any unwritten data for the stream to be delivered to the host environment.

For streams on which the program was just reading, `fflush()` discards any unread data in the buffers associated with the stream.

If `stream` is a NULL pointer, `fflush()` flushes all streams on which there is still unwritten data, that is, on all write streams.

RETURNS

0 (zero), if successful.

EOF, if an error occurred while trying to flush the buffer.

fgetc

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fgetc(FILE *stream);
```

DESCRIPTION

`fgetc()` reads in the next character from the given stream.

`fgetc()` returns an unsigned integer in the range 0 - 255 (decimal), or EOF.

Since `fgetc()` returns the integer value of the character, it is useful for getting bytes from binary files.

To read in multiple characters more efficiently, you can also use `fread()` or `fgets()`. To read text and data directly into variables, use `scanf()`.

RETURNS

The next character from the input stream, if successful.

EOF, if failure. Also, it sets file's error indicator if there was a read error, and sets the file's EOF indicator if it reaches the end of the file.

SEE ALSO

`fread()`, `fgets()`, `fscanf()`, `getc()`, `gets()`, `read()`, `scanf()`

fgetpos

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

DESCRIPTION

`fgetpos()` stores in the object that `pos` points to the current value of the file position indicator for the stream that `stream` points to.

RETURNS

Zero, if successful.

Non-zero, if failure. It also stores a positive value in `errno`.

fgets

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

DESCRIPTION

`fgets()` tries to read the next `n-1` characters from the given stream into the array `s`.

`fgets()` stops reading after it encounters a new-line character or after EOF, stores the new-line followed by a NULL character into `s`, and returns successful.

`fgets()` writes a NULL character immediately after the last character it writes to `s`.

RETURNS

`s`, if successful.

NULL, if failure. If it encounters an EOF and has not read any characters, it doesn't change the contents of `s` and sets the EOF error flag in the file descriptor. If a read error occurs, the contents of `s` are indeterminate.

floor

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double floor(double x);
```

DESCRIPTION

`floor()` rounds `x` down to the next lowest integer. In other words, `floor()` computes the largest integral value not larger than `x`. For example, `floor(1.1)` is `1.0`, and `floor(4.9)` is `4.0`.

RETURNS

The largest integral value not larger than `x`, as a double

fmod

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double fmod(double x, double y);
```

DESCRIPTION

fmod() computes the floating point remainder of x/y .

RETURNS

The remainder of x/y with the same sign as x , as a double, if successful
0 (zero), if y is zero. It also sets `errno` to `EDOM`.

fopen

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
FILE *fopen(char *filename, char *mode);
```

DESCRIPTION

`fopen()` opens the given file. `fopen()` automatically allocates a buffer for the file's stream, creates a new stream, and sets the stream to be fully-buffered. `fopen()` clears the error and EOF indicators for the stream.

These are the arguments to `fopen()`:

<code>filename</code>	pointer to the name of the file.
<code>mode</code>	pointer to a string describing how to open the file.

The three main modes are read, write, and append. Each mode has a similar set of variations: read binary, write binary, append binary, and so on.

These are all the possible modes:

String	Mode
<code>r</code>	Open this text file for reading.
<code>w</code>	Truncate this text file to zero length, or create a new text file for writing.
<code>a</code>	Append to this file. That is, open or create this text file for writing at end-of-file.
<code>rb</code>	Open this binary file for reading.
<code>wb</code>	Truncate this binary file to zero length, or create a new binary file for writing.
<code>ab</code>	Append to this binary file. That is, open or create this binary file for writing at end-of-file.
<code>r+</code>	Open this text file for updating (reading and writing).
<code>w+</code>	Truncate this text file to zero length or create a new text file for updating (reading and writing).
<code>a+</code>	Open this text file for updating (reading and writing), but append all writing at the end of the file.
<code>r+b</code> or <code>rb+</code>	Open this binary file for updating (reading and writing).
<code>w+b</code> or <code>wb+</code>	Truncate this binary file to zero length or create a new binary file for updating.

`a+b` or `ab+` Open this binary file for updating (reading and writing), but append all writing at the end of the file.

When opening a file for reading (i.e., mode is `r`, `rb`, `r+`, or `r+b/rb+`) and there is no file named `filename`, `fopen()` fails and returns `NULL`.

When opening a file for appending (i.e., mode is `a`, `ab`, `a+`, `a+b`, or `ab+`), all writing is appended to the end of the file, regardless of intervening calls to `fseek()`.

When opening a file for updating (i.e., `+` is the second or third character in mode), the program can both read from and write to the associated stream. However, between a read and a write (or vice versa), the program must make an intervening call to `fflush()` or one of the file positioning functions (`fseek()`, `fsetpos()`, or `rewind()`).

RETURNS

A pointer to the stream, if successful

`NULL`, if failure.

SEE ALSO

`open()`, `fileno()`, `fclose()`, `fflush()`, `fread()`, `fwrite()`.

`freopen()` opens a file on a specific stream.

fprintf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fprintf(FILE *stream, char *format, ...)
```

DESCRIPTION

`fprintf()` writes to the given stream. It takes the output from the data arguments, converts them according to the format specifiers in the format argument, and writes the result to the stream that `stream` points to.

These are the arguments to `fprintf()`:

<code>stream</code>	The stream <code>fprintf()</code> writes to.
<code>format</code>	Pointer to a string of format specifiers.
<code>...</code>	Represents the list of data that <code>fprintf()</code> writes.

For complete information on format specifiers, read the “`printf()` / `scanf()`” appendix

The data `fprintf()` prints must consist of valid expressions

RETURNS

The number of characters printed, if successful.

A negative value, if failure.

SEE ALSO

`fscanf()`, `printf()`, `scanf()`, `sprintf()`, `sscanf()`, `vfprintf()`, `vprintf()`, `vsprintf()`

fputc

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

DESCRIPTION

`fputc()` adds a single character `c` to the output stream `stream`.

`fputc()` writes the character at the position indicated by the file position indicator that is associated with the stream. It then advances the indicator appropriately.

If the file cannot support positioning requests, or if the stream was opened with append mode (see `fopen()`), `fputc()` appends the character to the end of the output stream.

RETURNS

The integer value of the character `c`, if successful

EOF, if failure (e.g., file is read only).

SEE ALSO

`putc()`, `fputs()`, `fwrite()`, `fprintf()`.

fputs

LIBRARY

ANSI

SYNTAX

```
#include <stdio>
int fputs(char *s, FILE *stream);
```

DESCRIPTION

`fputs()` writes the string to which `s` points to the stream to which `stream` points.

The string must end in a NULL (`'\0'`), but `fputs()` does not write the NULL to the stream.

RETURNS

0 (zero), if successful.

EOF, if failure

SEE ALSO

`puts()`, `fprintf()`, `fwrite()`, `fputc()`.

fread

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

DESCRIPTION

`fread()` reads items from the given stream and stores them in a block of memory.

`fread()` advances the file position indicator for the stream according to the number of characters it successfully reads.

If `fread()` reaches the end of the file before it has read the specified number of items, it sets the stream's EOF indicator and returns the number of items it was able to read.

These are the arguments to `fread()`:

<code>ptr</code>	Pointer to the area of memory into which <code>fread()</code> will store what it reads.
<code>size</code>	Size (uniform) of the items that <code>fread()</code> should read.
<code>nmemb</code>	Number of items that <code>fread()</code> should read.
<code>stream</code>	Stream from which <code>fread()</code> should read the items.

RETURNS

The number of items successfully read and stored, if successful.

The number of items successfully read and stored, if failure (e.g., read error) or end-of-file.

0 (zero), if `size` or `nmemb` is 0 (zero). But the contents of the area of memory into which it was supposed to read and the state of the stream remain unchanged.

free

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void free(void *ptr);
```

DESCRIPTION

`free()` releases the block of memory that `ptr` points to, making the memory available for further allocation.

Note: Make sure the argument `free()` receives is a pointer returned earlier by `calloc()`, `malloc()`, or `realloc()`. Otherwise, the results are unpredictable.

`free()` keeps its own internal accounting of how much memory to free.

If `ptr` is a `NULL` pointer, `free()` performs no action.

SEE ALSO

`calloc()`, `malloc()`, `cfree()`.

freopen

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
FILE *freopen(char *filename, char *mode,
               FILE *stream);
```

DESCRIPTION

`freopen()` opens a file using a specific stream, as opposed to `fopen()`, which opens a file on a new stream.

`freopen()` first tries to close any file that is associated with the stream `stream`.

If `freopen()` fails to close such a file successfully, it ignores the failure.

`freopen()` then opens the given file whose name is in the string that `filename` points to, and associates the stream `stream` with that file.

`freopen()` then clears the stream's error and EOF indicators.

RETURNS

The value of `stream`, if successful.

`NULL`, if the open fails.

SEE ALSO

`fopen()`, `fclose()`, `freopenc()` (in the console library).

frexp

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double frexp(double value, int *exp);
```

DESCRIPTION

`frexp()` breaks the floating-point number `value` into a normalized fraction (the mantissa) and an integral power of two (the exponent), so that

$$\text{mantissa} * 2^{(\text{exponent})} == \text{value}.$$

`frexp()` stores the exponent in `*exp`, and it returns the mantissa.

The mantissa is in the range 0.5 to 1 (or zero, if there's an error).

RETURNS

The mantissa. If `value` is 0 (zero), both the mantissa and the exponent will be zero.

SEE ALSO

`ldexp()`, the inverse function of `frexp()`.

fscanf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fscanf(FILE *stream, char *format, ...);
```

DESCRIPTION

`fscanf()` reads from the given stream. It separates the input according to the format specifiers in the format argument and stores the results in the objects pointed to by the data arguments.

`fscanf()` is the input analog of `fprintf()`. If the format is exhausted while arguments remain, `fscanf()` will evaluate the excess arguments but will otherwise ignore them.

These are the arguments to `fscanf()`:

<code>stream</code>	The stream from which <code>fscanf()</code> should read.
<code>format</code>	Pointer to a string of format specifiers.
<code>...</code>	Represents the items into which <code>fscanf()</code> will store the data it reads. These items must all be pointers.

For complete information on format specifiers for `fscanf()`, read the appendix on “`printf()` / `scanf()`.”

RETURNS

The number of items successfully assigned, if successful. This can be fewer than the number of items in its argument list or even zero in the event of an early matching failure (i.e., when an input argument does not match what `fscanf()` is looking for).

EOF, if an input failure occurs before it performs any data conversion.

SEE ALSO

`fprintf()`, `printf()`, `scanf()`, `sprintf()`, `sscanf()`,
`vfprintf()`, `vprintf()`, `vsprintf()`

fseek

LIBRARY

ANSI

SYNTAX

```
include <stdio.h>
int fseek(FILE *stream, long int offset,
          int whence);
```

DESCRIPTION

`fseek()` sets the file position indicator for the given stream. (The file position indicator measures the position in characters from the beginning of the file.)

When `fseek()` is successful, it clears the stream's EOF indicator and undoes any effects of `ungetc()` on the same stream.

`fseek()` is useful for performing nonsequential I/O to a file, since it lets the program manipulate the file position indicator itself rather than having to follow the indicator, which is set automatically by other I/O calls, through the file.

These are the arguments to `fseek()`:

<code>stream</code>	The file in which to seek.
<code>offset</code>	The amount of space in the file through which to seek.
<code>whence</code>	The position from which to start seeking.

These macros are useful values for `whence`:

<code>SEEK_SET</code>	The beginning of the file.
<code>SEEK_CUR</code>	The current value of the file position indicator.
<code>SEEK_END</code>	The end of the file.

For a binary stream, the new position will be `offset + whence`.

For a text stream, one of these must be true:

- `offset` is zero
- `offset` is a value returned by `ftell()` (on the same stream) and `whence` is `SEEK_SET`.

RETURNS

Zero, if successful.

Non-zero, if failure.

SEE ALSO

`lseek()` (in the Unix library).

fsetpos

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fsetpos(FILE *stream, fpos_t *pos);
```

DESCRIPTION

`fsetpos()` sets the file position indicator for the file that `stream` points to according to the setting of `pos`.

`pos` must be a value from an earlier call to `fgetpos()` (on the same stream).

A successful call to `fsetpos()` clears the EOF indicator for the stream and undoes any effects of `ungetc()` on the stream. After an `fsetpos()` call, the next operation on an update stream may be either read or write.

RETURNS

Zero, if successful.

Non-zero, if failure. It also stores a positive value in `errno`.

ftell

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
long int ftell(FILE *stream);
```

DESCRIPTION

`ftell()` obtains the current value of the file position indicator for the given stream.

The `fseek()` function can use this value to return the indicator to its position at the time of the `ftell()` call.

RETURNS

The current value of the file position indicator for `stream`, if successful.

`-1L`, if failure. It also stores a positive value in `errno`.

SEE ALSO

`tell()` (in the Unix library), `fseek()`

fwrite

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
size_t fwrite(void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

DESCRIPTION

`fwrite()` writes a block of data to `stream`.

These are the arguments to `fwrite()`:

<code>ptr</code>	Pointer to the memory to write to.
<code>size</code>	Size (uniform) of the items to write.
<code>nmemb</code>	Number of items to write.
<code>stream</code>	Stream to write to.

If `fseek()` has set the stream's file position indicator, `fwrite()` will advance the indicator by the number of characters `fwrite()` has successfully written.

RETURNS

The number of items successfully written, if successful.

The number of items successfully written before the failure, if failure.

SEE ALSO

`fread()`, `write()`, `fprintf()`, `fputs()`, `fputc()`, `putc()`.

getc

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int getc(FILE *stream)
```

DESCRIPTION

`getc()` is a macro that calls `fgetc()`.

Note: Since `getc()` is a macro and not a function, it may evaluate `stream` more than once. Make sure `stream` is not an expression that can have side effects.

RETURNS

The integer value of the next character from `stream`, if successful.

EOF, on error or end-of-file.

getchar

LIBRARY

ANSI

SYNTAX

```
#include <stdio>
int  getchar(void)
```

DESCRIPTION

`getchar()` is a macro that calls `fgetc()`, supplying `stdin` as the argument. This gets the next character from the standard input.

RETURNS

The integer value of the character, if successful.

EOF, if error or end-of-file.

getenv

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
char *getenv(char *name);
```

DESCRIPTION

getenv () is provided for ANSI and Unix compatibility. It always returns NULL.

RETURNS

NULL

gmtime

LIBRARY

ANSI

SYNTAX

```
#include <time.h>
struct tm *gmtime(time_t *timer);
```

DESCRIPTION

gmtime() is provided for ANSI and Unix compatibility. It always returns NULL.

RETURNS

NULL

isxxxx

SYNTAX

```
#include <ctype.h>
int isxxx(int c);
```

DESCRIPTION

isxxx() represents a family of functions that perform various tests on a character. These functions all work the same way: the program passes the function a single character as an argument, which the function tests. The function returns 0 (true) or non-zero (false) depending on whether the character c passes the test.

Function	Test
isalnum(c)	Is c an alphanumeric character? (a–z, A–Z, 0–9)
isalpha(c)	Is c an alphabetic character? (a–z, A–Z)
isctrl(c)	Is c a control character? (ASCII codes 0–1F inclusive and 7F, hexadecimal)
isdigit(c)	Is c a decimal digit? (0–9)
isgraph(c)	Is c a printing character (but not a space (' '))? (ASCII codes 21–7E inclusive, hexadecimal)
islower(c)	Is c a lowercase character? (a–z)
isprint(c)	Is c a printing character, including space (' ')? (ASCII codes 20–7E inclusive but not 7F, hexadecimal)
ispunct(c)	Is c a printing character that is neither space (' ') nor a character for which isalnum() is true? These are the punctuation characters: # \$ % * () " ' \ ` ~ : ; , < > / ? \ [] { } - _ = +
isspace(c)	Is c a standard white-space character? These are the whitespace characters: space (' ') form feed ('\f') new-line ('\n') carriage return ('\r') horizontal tab ('\t') vertical tab ('\v')
isupper(c)	Is c an uppercase character? (A–Z)
isxdigit(c)	Is c a hexadecimal digit? (0–9, A–F, a–f)

RETURNS

Non-zero, if the condition is true

Zero, if the condition is false.

labs

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
long int labs(long int j);
```

DESCRIPTION

`labs()` computes the absolute value of its long integer argument `j`.

RETURNS

`| j |` for any integer `j`, as a long integer.

ldexp

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double ldexp(double x, int exp);
```

DESCRIPTION

ldexp() multiplies the floating point number *x* by the integral power *exp* of 2.

RETURNS

$x * 2^{\text{exp}}$

SEE ALSO

frexp(), the inverse function of ldexp().

ldiv

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

DESCRIPTION

`ldiv()` is similar to `div()` except that the arguments and the members of the structure that is returned all have type `long int`.

It returns a structure of type `ldiv_t`, which looks like this:

```
struct {
    long int quot; /* quotient */
    long int rem;  /* remainder */
}
```

RETURNS

A structure containing the results of the division.

localeconv

LIBRARY

ANSI

SYNTAX

```
#include <locale.h>
struct lconv *localeconv(void);
```

DESCRIPTION

`localeconv()` is provided for ANSI C compatibility only. THINK C supports only the minimal locale environment. According to the standard, `localeconv()` returns an object of type `struct lconv`, with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

In THINK C, `localeconv()` returns values for the "C" locale, the minimal environment.

RETURNS

A pointer to an object of type `struct lconv`.

localtime

LIBRARY

ANSI

SYNTAX

```
#include <time.h>
struct tm *localtime(time_t *timer);
```

DESCRIPTION

`localtime()` converts the calendar time that `timer` points to to a broken-down time, expressed as local time.

RETURNS

A pointer to the broken-down time.

log

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double log(double x)
```

DESCRIPTION

log () computes the natural logarithm of x.

RETURNS

ln x (the natural logarithm of x), if x is greater than 0 (zero).

Negative infinity, if x is zero or negative.

log10

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double log10(double x)
```

DESCRIPTION

log10 () computes the base-10 logarithm of x.

RETURNS

log x (the base-10 logarithm of x) , if x is greater than 0 (zero).

Negative infinity, if x is zero or negative.

longjmp

LIBRARY

ANSI

SYNTAX

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

DESCRIPTION

`longjmp()` together with `setjmp()` lets you do a non-local goto.

`longjmp()` bypasses the usual function call and return mechanisms, and jumps directly back to the point established by a `setjmp()` call. `setjmp()` sets up the buffer that `longjmp()` specifies in the `env` argument.

`longjmp()` restores the environment that was saved by the most recent call to `setjmp()`.

Don't use `longjmp()` if you haven't previously called `setjmp()` or if you called `setjmp()` in a function that has terminated execution.

When `longjmp()` is called, all accessible objects have values, but some values are indeterminate. An object will have an indeterminate value only if it satisfies these qualifications :

- It is of automatic storage duration.
- It is local to the function that contains the corresponding `setjmp()` call.
- It was changed between the call to `setjmp()` and the call to `longjmp()`.

RETURNS

`val`

(After `longjmp()` completes, the program continues executing as if the corresponding call to `setjmp()` had just returned `val`. The only exception is that since `setjmp()` uses 0 (zero) to denote a true return, if `val = 0`, `setjmp()` converts 0 to 1 and returns 1, to continue to consistently denote that the return is from `longjmp()`.)

SEE ALSO

`setjmp()`

malloc

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void *malloc(size_t size);
```

DESCRIPTION

`malloc()` allocates a block of memory for an object. `size` specifies the size of the object.

Unlike `calloc()`, `malloc()` does not clear the block of memory.

To free the memory that `malloc()` allocates, use `free()`.

RETURNS

A pointer to the block of space, if successful.

NULL, if failure

SEE ALSO

`free()`, `calloc()`, `realloc()`.

mblen

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
int mblen(char *s, size_t n);
```

DESCRIPTION

`mblen()` is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, `mblen()` determines the number of bytes comprising the multi-byte character `s` points to.

In THINK C, `mblen()` returns 0 if `s` is NULL; 1, otherwise.

RETURNS

1, if `s` is not NULL.

0, if `s` is NULL

mbstowcs

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, char *s,
                size_t n);
```

DESCRIPTION

`mbstowcs()` is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, `mbstowcs()` converts a multi-byte character to which `s` points to a sequence of codes, stores it in the array `pwcs` points to, and returns the number of array elements changed.

In THINK C, `wcstombs()` simply copies the elements of `s` to `pwcs`, stopping after it copies `n` elements or a NULL character.

RETURNS

The number of characters copied.

SEE ALSO

`mblen()`, `wcstombs()`

mbtowc

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, char *s, size_t n);
```

DESCRIPTION

`mbtowc()` is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, `mbtowc()` converts the multi-byte character to which `s` points to a code (converting no more than `n` bytes), stores the code in the object `pwc` points to, and returns the length in bytes of the multi-byte character.

In THINK C, `wctomb()` sets `wchar` to the first element of `s`, if `s` is not `NULL`, and `n` is not 0.

RETURNS

1, if successful.

0, if `s` is `NULL`.

-1, if `n` is 0.

SEE ALSO

`mblen()`, `wctomb()`

memchr

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
void *memchr(void *s, int c, size_t n);
```

DESCRIPTION

`memchr()` locates the first occurrence of `c` in the initial `n` characters of the object that `s` points to.

The program should make sure that `c` is in the range 0 - 255 inclusive, decimal.

RETURNS

A pointer to the located character, if it finds the character.

NULL, if it doesn't find the character.

memcmp

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
int memcmp(void *s1, void *s2,
           size_t n);
```

DESCRIPTION

memcmp () compares the first n characters of the object that s1 points to to the first n characters of the object that s2.

RETURNS

a positive integer	if s1's object is greater in value than s2's.
zero	if s1's object is equal to s2's.
a negative integer	if s1's object is less in value than s2's.

memcpy

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
void *memcpy(void *s1, void *s2, size_t n);
```

DESCRIPTION

`memcpy()` copies `n` characters from the object that `s2` points to into the object that `s1` points to.

The program must make sure that `s1` and `s2` do not overlap. If they might overlap, the program should use `memmove()` instead.

RETURNS

The value of `s1`.

memmove

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
void *memmove(void *s1, void *s2, size_t n);
```

DESCRIPTION

`memmove()` copies `n` character from the object that `s2` points to into the object that `s1` points to.

`memmove()` works correctly even if `s1` and `s2` overlap.

RETURNS

The value of `s1`.

memset

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

DESCRIPTION

memset () copies the value of c into each of the first n characters of the object that s points to.

memset () converts c to an unsigned char to perform the copying.

RETURNS

memset () returns the value of s.

mktime

LIBRARY

ANSI

SYNTAX

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

DESCRIPTION

`mktime()` converts a broken-down time into a calendar time. (To convert a calendar time into a broken-down time, use `ctime()`.)

This calendar time has the same encoding as that of the values that `time()` returns.

`mktime()` ignores the original values of the `tm_wday` and `tm_yday` components of the structure. The original values of the other components are not restricted to the ranges indicated

On successful completion, `mktime()` sets the values of the `tm_wday` and `tm_yday` components, and sets the other components to represent the specified calendar time, but it forces their values to the ranges indicated

`mktime()` does not set the final value of `tm_mday` until it determines `tm_mon` and `tm_year`.

RETURNS

The calendar time encoded as a value of type `time_t`, if successful
(`time_t`) - 1, if the calendar time cannot be represented.

modf

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double modf(double x, double *ip);
```

DESCRIPTION

`modf()` splits `x` into integral and fractional parts; both parts carry the same sign as `x`. `modf()` stores the integral part in `*ip` and returns the signed fractional part.

RETURNS

The signed fractional part of `x`.

perror

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
void perror(char *s);
```

DESCRIPTION

`perror()` maps the error number in `errno` to string and prints an error message.

`perror()` writes this to `stderr`:

userString: errorMessage

where *userString* is the argument `s` and *errorMessage* is an error message string appropriate for the error number in `errno`.

The content of the error message string is the same as the one `strerror()` returns when the program hands `strerror() errno`.

pow

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double pow(double x, double y);
```

DESCRIPTION

`pow()` computes x^y , x raised to the power of y .

RETURNS

x^y , if x is greater than zero

Negative infinity, if x is zero and y is less than or equal to zero, or if x is negative and y is non-integral. It also sets `errno` to `EDOM`.

Infinity with the same sign as x , if the result is too large to represent.

printf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int printf(char *format, ...);
```

DESCRIPTION

`printf()` performs the same operations as `fprintf()` except that `printf()` writes its output to `stdout`.

For complete information on the format specifier arguments, read the “`printf()` / `scanf()`” appendix.

RETURNS

The number of characters it wrote to `stdout`, if successful

A negative value, if failure.

SEE ALSO

`fprintf()`, `fscanf()`, `scanf()`

putc

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int putc(char c, FILE *stream);
```

DESCRIPTION

putc () writes a character to the output stream.

putc () is similar to fputc () except that putc () is a macro and not a true function.

Note: Since putc () is a macro, it may evaluate stream more than once. Make sure the stream argument is not an expression that can have side effects.

RETURNS

The integer value of the character, if successful.

EOF, if failure. It also sets the error indicator for the stream.

SEE ALSO

fputc (),getc (),fgetc ()

putchar

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int putchar(int c);
```

DESCRIPTION

`putchar()` calls the `fputc()` function, supplying `stdout` as the output stream.

`putchar()` is implemented as a macro rather than a true function.

RETURNS

The character written, if successful.

EOF, if failure. It also sets the error indicator for the stream.

SEE ALSO

`putc()`, `puts()`, `printf()`

puts

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int puts(char *s);
```

DESCRIPTION

`puts()` writes the given string to `stdout`, appending a newline (`'\n'`) character to the output.

`s` points to the string. The string must have a `NULL` character as a terminator; but `puts()` does not write this terminator to the output.

RETURNS

0 (zero), if successful

EOF, if failure

SEE ALSO

`fputs()`, `printf()`, `putchar()`.

qsort

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar) (void *, void *));
```

DESCRIPTION

qsort () sorts an array in ascending order according to the results of a comparison function.

These are the arguments to qsort ():

base	Pointer to the initial element of the array.
nmemb	Number of elements in the array.
size	Size of each element in the array.
compar	The comparison function, which takes two arguments. The arguments should be pointers to the objects it compares.

The comparison function returns an integer less than, equal to, or greater than zero according to whether it considers the first object less than, equal to, or greater than the second argument, respectively.

raise

LIBRARY

ANSI

SYNTAX

```
#include <signal.h>
int raise(int sig);
```

DESCRIPTION

`raise()` sends the signal `sig`.

The signal-handling software then processes the signal according to the settings of the program's most recent call to `signal()`.

`signal()` can specify that the signal-handling software ignore the signal, use default signal handling, or dispatch to a specific (e.g., user-written) signal-handling routine.

For more information on signal-handling, see the section on `signal()`.

RETURNS

Zero, if successful.

Non-zero, if failure.

SEE ALSO

`signal()`, `kill()` (in the Unix library)

rand

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
int rand(void);
```

DESCRIPTION

rand() returns a pseudo-random integer in the range 0 to RAND_MAX.

Successive calls to rand() result in a pseudo-random sequence of numbers.

If you set the seed (using srand()) to the same number each time you run your program, you'll always get the same sequence of pseudo-random numbers.

RETURNS

A pseudo-random integer in the range 0 to RAND_MAX.

SEE ALSO

srand().

realloc

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

`realloc()` changes the size of a given memory region while preserving the region's contents. If necessary, `realloc()` copies the contents to another region of memory.

`realloc()` is useful when the program needs a larger size specification (e.g., long rather than int).

These are the arguments to `realloc()`:

<code>ptr</code>	A pointer that was returned earlier by a call to <code>calloc()</code> , <code>malloc()</code> , or <code>realloc()</code> .
<code>size</code>	The desired new size of the memory region.

Note: If `ptr` points to space that has been deallocated (e.g., it matches a pointer that was returned by a call to `free()` or `realloc()`), the behavior of `realloc()` is undefined.

`realloc()` handles these special cases:

- If `ptr` is NULL, `realloc()` behaves like a call to `malloc()` for the specified size.
- If the new size is smaller than the old size, `realloc()` takes away space at the end of the old region and discards the contents of that space.
- If `size` is 0 (zero) and `ptr` is not NULL, `realloc()` frees the region that `ptr` points to.
- If the new size cannot be allocated, `realloc()` leaves the region unchanged and returns NULL.

RETURNS

A pointer to the new memory space, if successful.

NULL, if failure.

SEE ALSO

`malloc()`.

remove

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int remove(char *filename);
```

DESCRIPTION

remove () deletes a file.

If the file is open, remove () returns nonzero and the file remains unchanged.

RETURNS

Zero, if successful.

Non-zero, if failure.

rename

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int rename(char *old, char *new);
```

DESCRIPTION

`rename ()` changes the name of a file from the string that `old` points to to the string that `new` points to.

RETURNS

Zero, if successful.

Non-zero, if failure. If the file existed previously, its name remains unchanged.

rewind

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
void rewind(FILE *stream);
```

DESCRIPTION

`rewind()` sets the file position indicator for given stream to the beginning of the file.

Remember that this indicator is moved automatically during I/O and it is moved explicitly by `fseek()`.

`rewind()` also clears the error indicator for the stream.

SEE ALSO

`fseek()`, `lseek()`

scanf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int scanf(char *format, ...);
```

DESCRIPTION

`scanf()` performs the same operations as `fscanf()` except that `scanf()` uses `stdin` as the input stream.

For complete information on format specifiers, read the appendix on “`printf()` / `scanf()`.”

RETURNS

The number of items successfully read, if successful. This can be fewer than the number of items in the argument list or even zero in the event of an early matching failure, i.e., when an input argument does not match what `scanf()` is looking for.

EOF, if an input failure occurs before any data conversion.

setbuf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

DESCRIPTION

Except that `setbuf()` returns no value, `setbuf()` is similar to `setvbuf()`, as follows:

- When `buf` is not `NULL`, `setbuf()` is the same as calling `setvbuf()` with `_IOFBF` for mode and `BUFSIZ` for size.
- When `buf` is `NULL`, `setbuf()` is the same as calling `setvbuf()` with `_IONBUF` for mode.

setjmp

LIBRARY

ANSI

SYNTAX

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

DESCRIPTION

setjmp() together with longjmp() allows the program to do a non-local goto. This is useful for returning error information or other information that might otherwise get lost.

setjmp() establishes a destination to which a subsequent longjmp() call can jump. On this invocation of setjmp(), setjmp() returns 0 (zero).

longjmp() bypasses the usual function call and return mechanisms and jumps back to the point just after the setjmp() call as if setjmp() itself had just returned. On this return, setjmp() returns nonzero.

Remember, setjmp() returns 0 (zero) on true return, non-zero on a return from a longjmp() call. longjmp() itself never returns; instead, execution proceeds as if the setjmp() call had returned.

After a longjmp() occurs, variables in the function that called longjmp() are lost, and local non-static variables in the function that calls setjmp() are indeterminate

setjmp() saves the environment in the array env, for use by longjmp().

RETURNS

Zero, if returning from a direct invocation.

Non-zero, if returning from a call to longjmp().

1, if returning from a call a call to longjmp() with val = 0.

SEE ALSO

longjmp()

setlocale

LIBRARY

ANSI

SYNTAX

```
#include<locale.h>
char *setlocale(int category, char *locale);
```

DESCRIPTION

setlocale() is provided for ANSI C compatibility only. THINK C supports only the minimal "C" locale environment.

In THINK C, setlocale() returns "C" if locale is either "C" or NULL. Otherwise, it returns NULL.

RETURNS

"C", if locale is "C" or NULL.

NULL, otherwise.

setvbuf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

DESCRIPTION

`setvbuf()` lets the program specify how the given stream will be buffered, and it lets the program either supply a buffer or have `setvbuf()` automatically allocate a buffer.

You should use `setvbuf()` after the stream that `stream` points to has been associated with an open file and before any other operation is performed on that stream.

These are the arguments to `setvbuf()`:

<code>mode</code>	One of the following values, describing how the stream will be buffered: <code>_IOFBF</code> Input/output will be fully buffered. <code>_IOLBF</code> Input/output will be line buffered. <code>_IONBF</code> Input/output will be unbuffered.
<code>buf</code>	Pointer to a buffer that the program supplies. If <code>buf</code> is <code>NULL</code> , <code>setvbuf()</code> allocates a buffer itself.
<code>size</code>	The size of the buffer that <code>setvbuf()</code> should allocate. If the file is a disk file, <code>size</code> should be a multiple of 512.

RETURNS

Zero, if successful.

Non-zero, if failure. If `setvbuf()` is called with an invalid value for `mode`, or if `setvbuf()` cannot allocate a buffer, or if the buffer that `buf` points to is too small, `setvbuf()` will not honor the request and will set `mode` to `_IONBF` (unbuffered).

signal

LIBRARY

ANSI

SYNTAX

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

`signal()` is part of the signal mechanism, which consists of `signal()`, `raise()`, `kill()`, and the underlying signal-handling software. The signal mechanism lets a program respond to external events and send messages to itself.

Keep in mind that signals are not a balanced sender-receiver mechanism, despite the send-respond language used to describe them.

To send a signal, call `raise()` or `kill()`.

To specify the correct response to a signal, use `signal()`. `signal()` lets you specify ahead of time what the signal-handling software should do when a given signal occurs. You can change the response that `signal()` specifies many times before the signal occurs by simply calling `signal()` again with different values.

Once the signal occurs, the signal-handling software performs the response specified by the last `signal()` call made. The software also resets this response to be the default response.

After a given signal occurs, you must call `signal()` again in order to re-establish the non-default response for that signal. (You may leave the default setting as is if desired.)

At program start-up, the signal software will initially ignore no signal and will perform the default response for all signals.

Thereafter, `signal()` lets a program specify one of three responses — no response, default response, and user-defined response — as follows:

- If the value of `func` is `SIG_DFL`, the signal-handling software will perform default handling of that signal (in THINK C, an abnormal program termination.)

- If `func` points to a function, the signal-handling software calls the function, using `sig` as the argument (and it resets `func` to be the default function thereafter). This called function executes normally. a typical task is to set a flag and then return.
- If the value of `func` is `SIG_IGN`, the signal is ignored.

These are the arguments to `signal()`:

<code>sig</code>	The signal number of the signal. Valid signals are listed in the next section.
<code>func</code>	The desired outcome for the given signal. <code>func</code> can be a pointer to a function or one of the following values: <code>SIG_DFL</code> Perform default handling of the signal. <code>SIG_IGN</code> Ignore the signal. If <code>func</code> is a pointer to a function, that function cannot call any library routines.
<code>pointer</code>	Call the function that this pointer points to when the signal <code>sig</code> occurs.

Avoid the following situations:

- `func` uses `return`, and the value of `sig` is `SIG_FPE` or any other implementation-defined value that corresponds to a computational exception.
- The signal does not come from `abort()` or `raise()`, and either the signal handler calls any function in the standard library other than `signal()` or the signal handler refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `sig_atomic_t`. Furthermore, if such a call to `signal()` causes `signal()` to return unsuccessfully, the value of `errno` is indeterminate.

RETURNS

The value of `func` for the most recent call for the specified signal `sig`, if successful.

`SIG_ERR`, if failure. It also stores a positive value in `errno`.

SEE ALSO

`raise()`, `kill()` (in the Unix library)

sprintf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int sprintf(char *s, char *format, ...);
```

DESCRIPTION

`sprintf()` performs the same operations as `fprintf()` except that it writes its output to the string `s` points to, instead of a stream.

`sprintf()` appends a NULL character to the end of the characters it writes.

For complete information on format specifiers, read the appendix on “printf / scanf.”

RETURNS

The number of characters written to `s`, not counting the terminating NULL.

SEE ALSO

`fprintf()`, `printf()`, `scanf()`, `sscanf()`

srand

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
void srand(unsigned int seed);
```

DESCRIPTION

`srand()` initializes the pseudo-random number generator, using its argument as a seed for a new sequence.

Use `rand()` to produce a pseudo-random number.

Using the same number as the seed for `srand()` will always produce the same sequence of numbers.

sscanf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int sscanf(char *s, char *format, ...)
```

DESCRIPTION

`sscanf()` performs the same operations as `fscanf()` except that it reads its input from the string that `s` points to, instead of a stream.

Reaching the end of the string for `sscanf()` is the same as reaching the end of a file for `fscanf()`.

For complete information on format specifiers for `fscanf()`, read the appendix on “`printf()` / `scanf()`.”

RETURNS

The number of items read, if successful. This can be fewer than the number of items in the argument list or even zero in the event of an early matching failure, i.e., when an input argument does not match what `scanf()` is looking for.

EOF, if an input failure occurs before data conversion.

strcat

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strcat(char *s1, char *s2)
```

DESCRIPTION

`strcat()` appends a copy of the string that `s2` points to to the end of the string that `s1` points to.

The initial character of `s2` overwrites the NULL character at the end of `s1`.

Make sure there is enough space for string `s2` in the character array after the end of the string `s1`.

RETURNS

The value of `s1`, after `s2` has been appended.

strchr

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strchr(char *s, int c);
```

DESCRIPTION

`strchr()` locates the first occurrence of `c` (which it converts to a `char`) in the string that `s` points to.

`strchr()` considers the terminating `NULL` character to be part of the string `s`.

RETURNS

A pointer to the first occurrence of `c` in the string `s`, if `c` is found.

The `NULL` pointer, if `c` is not found.

SEE ALSO`strrchr()`

strcmp

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
int strcmp(char *s1, char *s2);
```

DESCRIPTION

`strcmp()` compares the string that `s1` points to to the string that `s2` points to.

Note that if `s1` is a substring of `s2`, `strcmp()` will return a number greater than zero because the last characters it compares will be the terminating NULL (`'\0'`) character of `s1` against some character in `s2`.

RETURNS

Positive integer	if <code>s1</code> is greater than <code>s2</code> .
0	if <code>s1</code> equals <code>s2</code> .
Negative integer	if <code>s1</code> is less than <code>s2</code> .

strcoll

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
int strcoll(char *s1, char *s2);
```

DESCRIPTION

`strcoll()` is provided for ANSI C compatibility only. THINK C supports only the minimal locale environment. According to the standard, `strcoll()` compares the string that `s1` points to to the string that `s2` points to, interpreting both strings as appropriate to the the current locale.

In THINK C, `strcoll()` returns the value that `strcmp()` would return for the same strings.

RETURNS

Positive integer	if <code>s1</code> is greater than <code>s2</code> .
0	if <code>s1</code> equals <code>s2</code> .
Negative integer	if <code>s1</code> is less than <code>s2</code> .

strcpy

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strcpy(char *s1, char *s2);
```

DESCRIPTION

`strcpy()` copies the string that `s2` points to (including the terminating NULL character (`'\0'`)) into the array that `s1` points to.

RETURNS

The value of `s1`.

SEE ALSO

`strncpy()`

strcspn

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
size_t strcspn(char *s1, char *s2);
```

DESCRIPTION

strcspn() computes the length of the maximum initial segment of the string that s1 points to that consists entirely of characters that are not from the string that s2 points to.

RETURNS

strcspn() returns the length of the segment.

strerror

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strerror(int errnum);
```

DESCRIPTION

`strerror()` uses the error number `errnum` to find the appropriate error message.

You should not be modify the string returned, but a subsequent call to the `strerror()` function may overwrite it.

RETURNS

A pointer to the string that contains the error message.

strftime

LIBRARY

ANSI

SYNTAX

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
                char *format, struct tm *timeptr);
```

DESCRIPTION

`strftime()` formats and prints a time.

If copying takes place between objects that overlap, the behavior is undefined.

These are the arguments to `strftime()`:

<code>s</code>	Pointer to the array into which <code>strftime()</code> should write the formatted time.
<code>maxsize</code>	Maximum number of characters that <code>strftime()</code> may place into the array <code>s</code> .
<code>format</code>	A string of one or more conversion specifiers, which <code>strftime()</code> replaces with actual values. The table below lists valid format specifiers and the values to which <code>strftime()</code> converts them.
<code>timeptr</code>	Pointer to a structure that contains the time that <code>strftime()</code> should format and write. It's the same one that <code>gmtime()</code> and <code>localtime()</code> return.

These are the specifiers for `strftime()`'s format argument:

Specifier	Replaced with:
<code>%a</code>	The abbreviated weekday name.
<code>%A</code>	The full weekday name.
<code>%b</code>	The abbreviated month name.
<code>%B</code>	The full month name.
<code>%c</code>	The appropriate date and time representation.
<code>%d</code>	The day and month as a decimal number (Range: 01–31).
<code>%H</code>	The hour (24-hour clock), as decimal number (Range: 00–23).
<code>%I</code>	The hour (12-hour clock), as a decimal number (Range: 01–12).
<code>%j</code>	The day of the year, as a decimal number (Range: 001–366).
<code>%m</code>	The month, as a decimal number (Range: 01–12).
<code>%M</code>	The minute, as a decimal number (Range: 00–59).
<code>%P</code>	The AM/PM designation for a 12-hour clock.
<code>%S</code>	The second, as a decimal number (Range: 00–61).
<code>%U</code>	The week number of the year (with Sunday as the first day of week 1), as a decimal number (Range: 00–53).
<code>%w</code>	The weekday, as a decimal number (Range: 0 (Sunday) – 6).
<code>%W</code>	The week number of the year (with Monday as the first day of week 1), as a decimal number (Range: 00–53).
<code>%x</code>	The locale's appropriate date representation.
<code>%X</code>	The locale's appropriate time representation.
<code>%y</code>	The year without century, as a decimal number (Range: 00–99).
<code>%Y</code>	The year with century, as a decimal number.
<code>%z</code>	The time zone name or abbreviation, or by no characters if no time zone is determinable.
<code>%%</code>	<code>%</code>

RETURNS

The number of characters that placed into the array `s`, not including the terminating `NULL`, if successful.

0 (zero), if the string generated, including the terminating `NULL`, is longer than `maxsize`. Also, the contents of the array are indeterminate.

strlen

LIBRARY

ANSI

SYNTAX

```
#include <strings.h>
size_t strlen(char *s);
```

DESCRIPTION

strlen() computes the length of the string that s points to.

RETURNS

strlen() returns the number of characters that precede the terminating NULL character.

strncat

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strncat(char *s1, char *s2, size_t n)
```

DESCRIPTION

`strncat()` appends a copy of the string that `s2` points to to the end of the string that `s1` points to, until it has appended `n` characters or it has reached the end of string `s2`.

The initial character of `s2` overwrites the NULL character at the end of `s1`.

If `strncat()` appends `n` characters without reaching the end of `s2`, `strncat()` will add a terminating NULL character (`'\0'`).

Make sure there is enough space for `s2` in the character array after the end of the string `s1`.

RETURNS

The value of `s1`, after appending the string `s2`. If `n` is negative or zero, it returns `s1` unchanged.

SEE ALSO`strcat()`

strncmp

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
int strncmp(char *s1, char *s2, size_t n);
```

DESCRIPTION

strncmp() compares the string that s1 points to to the string that s2 points to, up to a limit of n characters.

strncmp() does not compare characters that follow the NULL character.

RETURNS

Positive integer	if the first n characters of s1 are greater than the first n characters of s2.
0	if the first n characters of s1 are equal to the first n characters of s2.
Negative integer	if the first n characters of s1 are less than the first n characters of s2.

SEE ALSO

strcmp()

strncpy

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strncpy(char *s1, char *s2, size_t n);
```

DESCRIPTION

strncpy () copies characters from s2 to s1 until either it has copied n characters or it reaches a NULL character in s2.

If s2 is shorter than n characters, strncpy () will append NULL characters to s1 until it has written n characters there.

RETURNS

The value of s1.

strpbrk

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strpbrk(char *s1, char *s2);
```

DESCRIPTION

strpbrk() locates the first occurrence of any character from s2 in s1.

RETURNS

A pointer to the first occurrence of any character from s2 in s1, if such a character is found.

NULL, if no character from s2 is found in s1.

strrchr

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strrchr(char *s, int c);
```

DESCRIPTION

`strrchr()` locates the last occurrence of `c` (which it converts to a `char`) in the string that `s` points to.

`strrchr()` considers the terminating `NULL` character to be part of the string `s`.

RETURNS

A pointer to `c`'s last occurrence in the string `s`, if `c` is in `s`.

`NULL`, if `c` is not in `s`.

SEE ALSO

`strchr()`

strspn

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
size_t strspn(char *s1, char *s2)
```

DESCRIPTION

`strspn()` computes the length of the maximum initial segment of the string that `s1` points to that consists entirely of characters from the string that `s2` points to.

RETURNS

The length of the segment (i.e., the position of the first character in `s1` that is not in `s2`).

strstr

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strstr(char *s1, char *s2);
```

DESCRIPTION

`strstr()` looks for string `s2` in string `s1`. `strstr()` does not use the NULL terminator in `s2`, if any, in comparing the two strings.

RETURNS

A pointer to the found string, if string `s2` is in string `s1`.

NULL, if string `s2` is not in string `s1`.

`s1`, if `s2` points to a string with zero length.

strtod

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
double strtod(char *nptr, char **endptr);
```

DESCRIPTION

`strtod()` converts the initial portion of the given string to a double.

These are the arguments to `strtod()`:

<code>nptr</code>	Points to the string that <code>strtod()</code> should convert.
<code>endptr</code>	Pointer to a pointer to any unconverted suffix characters, unless <code>endptr</code> is NULL.

RETURNS

The converted value, if any.

Infinity matching the sign of the value, if the converted value is outside the range of representable values or would cause underflow. It also sets `errno` to `ERANGE`.

0 (zero), if it could not perform any conversion.

strtok

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
char *strtok(char *s1, char *s2);
```

DESCRIPTION

`strtok()` breaks a string into tokens. You must call `strtok()` a separate time for each token.

To have `strtok()` continue to work on the same string, call `strtok()` the first time with the `s1` argument non-NULL. Then call `strtok()` each succeeding time for that string with the `s1` argument set to NULL. (Setting the `s1` argument to NULL tells `strtok()` to continue working on the string it was working on at the previous invocation.)

If `strtok()` finds the beginning of a token but finds no further delimiter characters, the current token extends to the end of the string `s1`, and subsequent calls to `strtok()` for a token will return a NULL pointer.

When `strtok()` finds the end of a token (i.e., it finds a delimiter character from `s2` at the end of a token in `s1`) it overwrites the found delimiter in `s1` with a NULL character, which terminates the current token.

`strtok()` saves a pointer to the following character, from which `strtok()`'s next search for a token will start.

These are the arguments to `strtok()`:

`s1` points to the string to be broken up.

`s2` points to a string consisting of valid token delimiters. The string that `s2` points to may be different from call to call.

RETURNS

The first character of the token, if it finds a token.

NULL, if it does not find a token (i.e., if it does not find any characters in `s1` that are not in `s2`, then there are no tokens in string `s1`).

strtol

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
long int strtol(char *nptr, char **endptr,
                int base);
```

DESCRIPTION

`strtol()` converts a string to a long integer.

These are the arguments to `strtol()`:

<code>nptr</code>	Pointer to the string that <code>strtol()</code> should convert.
<code>endptr</code>	Pointer to a pointer to any unconverted suffix characters, unless <code>endptr</code> is <code>NULL</code> .
<code>base</code>	The base of the number into which <code>strtol()</code> should convert the given string.

If `base` is between 2 and 36, `strtol()` assumes the input is an integer in that base. If `base` is 0 (zero), `strtol()` assumes the input is an integer constant in base 8, 10, or 16. It determines the base of the number according to the prefix characters. A leading 0 implies the number is octal (base 8); a leading 0x or 0X implies the number is hexadecimal (base 16).

The string should have any number of (or no) white-space characters (such as spaces and tabs), followed by the integer, and then any number of (or no) unrecognized characters. The integer may be optionally preceded by a plus or minus sign. Any integer suffix (such as L or U) is ignored. The letters a-z and A-Z are assigned the values 10-35 and can be part of the integer if the value of the letter is less than `base` (e.g. base 16 integers can include the letters a-f and A-F, which are assigned the values 10-15). If `base` is 16, the integer may be optionally preceded by 0x or 0X.

If the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or permissible letter or digit, `strtol()` considers the subject sequence to contain no characters, and so it performs no conversion.

RETURNS

The converted value as a long integer, if any.

`LONG_MAX`, if the converted value is too large for the range of representable values. It also sets `errno` to `ERANGE`.

`LONG_MIN`, if the converted value is too small for the range of representable values. It also sets `errno` to `ERANGE`.

0, if it could not convert the string.

SEE ALSO

`strtoul()`

strtoul

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
unsigned long int strtoul(char *nptr,
                          char **endptr, int base);
```

DESCRIPTION

`strtoul()` converts a string to an unsigned long integer.

`strtoul()` follows essentially the same conversion rules that `strtoul()` does.

These are the arguments to `strtoul()`:

<code>nptr</code>	Pointer to the string that <code>strtoul()</code> should convert.
<code>endptr</code>	Pointer to a pointer to any unconverted suffix characters, unless <code>endptr</code> is <code>NULL</code> .
<code>base</code>	Specifies the base of the number into which <code>strtoul()</code> should convert the given string.

RETURNS

The converted value, if any.

`ULONG_MAX`, if the converted value is outside the range of representable values. It also sets `errno` to `ERANGE`.

0, if it could not convert the string.

SEE ALSO

`strtoul()`

strxfrm

LIBRARY

ANSI

SYNTAX

```
#include <string.h>
size_t strxfrm(char *s1, char *s2, size_t n);
```

DESCRIPTION

`strxfrm()` is provided for ANSI C compatibility only. According to the standard, it transforms the string that `s2` points to (preparing the string for use with `strcmp()`) and places the resulting string in the array that `s1` points to.

In THINK C, `strxfrm()` simply copies the string to which `s1` points to the array to which `s2` points, without performing any transformation.

If `n` is zero, `s1` can be a NULL pointer.

RETURNS

The length of the transformed string, not including the terminating NULL.

system

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
int system(char *string);
```

DESCRIPTION

`system()` is provided for ANSI C compatibility only. According to the standard, it passes the given string to the host environment for execution by a command processor.

Since the Macintosh has no command processor, THINK C ignores the command string and always returns 0 (zero).

RETURNS

0 (zero)

tan

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double tan(double x);
```

DESCRIPTION

`tan()` returns the tangent of `x`, measured in radians.

RETURNS

The tangent of `x`, measured in radians.

Infinity, if `x` is an odd multiple of $\pi/2$. It also sets `errno` to `EDOM`.

tanh

LIBRARY

ANSI

SYNTAX

```
#include <math.h>
double tanh(double x);
```

DESCRIPTION

`tanh()` returns the hyperbolic tangent of `x`.

RETURNS

The hyperbolic tangent of `x`.

time

LIBRARY

ANSI

SYNTAX

```
#include <time.h>
time_t time(time_t *timer);
```

DESCRIPTION

time() determines the current calendar time.

If timer is not a NULL pointer, time() assigns the return value to the object that timer points to.

RETURNS

The current calendar time.

(time_t) -1, if the calendar time is not available.

tmpfile

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
FILE *tmpfile(void)
```

DESCRIPTION

`tmpfile()` creates a temporary binary file for use by the the program.

`tmpfile()` uses write-binary-update mode (`wb+`, as described in the section on `fopen()`) to open the file for update. `tmpfile()` will automatically remove the file when the file is closed (e.g., with `fclose()`) or when the program terminates.

If the program terminates abnormally, the file is deleted.

RETURNS

A pointer to the stream of the file created.

`NULL`, if it could not create the file.

tmpnam

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
char *tmpnam(char *s);
```

DESCRIPTION

tmpnam() generates a string that is a valid filename and that is not the same as the name of an existing file.

tmpnam() generates a different string each time it is called, up to TMP_MAX times.

If the argument is a NULL pointer, tmpnam() creates a new filename, leaves it in an internal static structure, and returns a pointer to that structure.

If the argument is not a NULL pointer, tmpnam() creates a new filename, writes the new name into the structure the pointer points to, then returns the same pointer.

RETURNS

A pointer to a structure that contains the new filename.

tolower

LIBRARY

ANSI

SYNTAX

```
#include <ctype.h>
int tolower(int c);
```

DESCRIPTION

`tolower()` converts an uppercase letter to the corresponding lowercase letter.

RETURNS

The corresponding lowercase letter.

`c` unchanged, if `c` is not a character for which `isupper()` is true, or if there is no corresponding character for which `islower()` is true.

toupper

LIBRARY

ANSI

SYNTAX

```
#include <ctype.h>
int toupper(int c);
```

DESCRIPTION

`toupper()` converts a lowercase letter to the corresponding uppercase letter.

RETURNS

The corresponding uppercase letter.

`c` unchanged, if `c` is not a character for which `islower()` is true, or if there is no corresponding character for which `isupper()` is true

ungetc

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

DESCRIPTION

`ungetc()` pushes a single character `c` back onto the input stream `stream`.

The next read from that stream will get that character first. On the other hand, a successful intervening call to a file-positioning function (`fseek()`, `fsetpos()`, or `rewind()`) for the stream `stream` discards any pushed-back characters for that stream.

`ungetc()` converts the character to an unsigned char before pushing it back onto the input stream.

If the value of `c` equals EOF, `ungetc()` fails and the input stream is unchanged.

RETURNS

`c`, if successful.

EOF, if not successful.

va_start

va_arg

va_end

LIBRARY

ANSI

SYNTAX

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

DESCRIPTION

These three macros allow a function to step through an argument list when the list contains arguments of varying number and type.

The function declares its variable argument list as follows:

```
function (parmN, ...)
```

You must declare at least one named function argument, since `va_start()` uses the last named argument (*parmN* in the example above) to get started.

You must also declare within the function a variable of type `va_list`. This variable will point to each argument in turn:

```
va_list ap;
```

First, call `va_start()`:

```
va_start(ap, parmN);
```

You must call `va_start()` once before accessing the unnamed arguments. This initializes `ap` so that it points to the first argument after *parmN*. *parmN* stands for the last named argument (the one just before the “`, ...`”) in the function's definition.

For each unnamed argument, call `va_arg()`:

```
va_arg(ap, type);
```

Each call to `va_arg()` returns one argument and steps `ap` to the next. `va_arg()` uses the type name *type* to determine what type of argument to return and how big a step down the argument stack to take.

Lastly, call `va_end()`:

```
va_end(ap);
```

`va_end()` deallocates `ap`. This allows a normal return from the function. You must call `va_end()` once after processing all its arguments but before exiting.

`va_start()`, `va_arg()`, and `va_end()` are implemented as macros rather than actual functions.

Do not declare *parmN* as

- register storage class
- a function or array type
- a type that is not compatible with the type that results after application of the default argument promotions.

fprintf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int fprintf(FILE *stream, char *format,
            va_list arg);
```

DESCRIPTION

`fprintf()` performs the same operations as `fprintf()`, except that `fprintf()` takes a single argument instead of a variable argument list.

However, the single argument comes from a variable argument list. The calling function must make sure `arg` is a `va_list` pointer and has been initialized with the `va_start()` macro (and possibly subsequent `va_arg()` calls).

`fprintf()` does not invoke `va_end()`. The calling function must make sure to do so when appropriate.

For complete information on format specifiers, read the appendix on “printf / scanf.”

RETURNS

The number of characters written, if successful

A negative value, if failure.

SEE ALSO

`fprintf()`, `fscanf()`, `vprintf()`, `vsprintf()`

vprintf

LIBRARY

ANSI

SYNTAX

```
#include <stdio>
int vprintf(char *format, va_list arg);
```

DESCRIPTION

`vprintf()` performs the same operations as `fprintf()`, except that `vprintf()` takes a single argument instead of a variable argument list.

However, the single argument comes from a variable argument list. The calling function must make sure `arg` is a `va_list` pointer and has been initialized with the `va_start()` macro (and possibly subsequent `va_arg()` calls).

`vprintf()` does not invoke `va_end()`, so the calling function must make sure to do so when appropriate.

For complete information on format specifiers, read the appendix on “`printf()` / `scanf()`.”

RETURNS

The number of characters written, if successful.

A negative value, if failure.

vsprintf

LIBRARY

ANSI

SYNTAX

```
#include <stdio.h>
int vsprintf(char *s, char *format,
             va_list arg);
```

DESCRIPTION

`vsprintf()` performs the same operations as `sprintf()`, except that `vsprintf()` takes a single argument instead of a variable argument list.

However, the single argument comes from a variable argument list. The calling function must make sure `arg` is a `va_list` pointer and has been initialized with the `va_start()` macro (and possibly subsequent `va_arg()` calls).

`vsprintf()` does not invoke `va_end()`, so the calling function must make sure to do so when appropriate.

For complete information on format specifiers, read the appendix on “`printf()` / `scanf()`.”

RETURNS

The number of characters it writes, not counting the terminating NULL character, if successful.

wctomb

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

DESCRIPTION

wctomb() is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, wctomb() converts wchar to a multi-byte character, stores it in the array s points to, and returns the length in bytes of the multi-byte character.

In THINK C, wctomb() sets the first element of s to wchar, if s is not a NULL pointer.

RETURNS

1, if s is not a NULL pointer.

0, if s is a NULL pointer.

SEE ALSO

mblen(), mbtowc()

wcstombs

LIBRARY

ANSI

SYNTAX

```
#include <stdlib.h>
size_t wcstombs(wchar_t *pwcs, char *s,
                size_t n);
```

DESCRIPTION

`wcstombs()` is provided for ANSI C compatibility only. THINK C does not support multi-byte characters. According to the standard, `wcstombs()` converts a sequence of codes to a multi-byte character string, stores it in the array `s` points to, and returns the number of array elements changed.

In THINK C, `wcstombs()` simply copies the elements of `pwcs` to `s`, stopping after it copies `n` elements or a NULL character.

RETURNS

The number of characters copied.

SEE ALSO

`mblen()`, `mbstowcs()`

console 4

Introduction

This chapter describes THINK C's `console` library and includes a complete listing of all the functions it contains.

Topics covered in this chapter:

- What is the console library?
- What is a console?
- Environments
- Options
- A catalogue of functions

What Is the Console Library?

The console library makes available to programs a group of routines that perform certain console operations, such as creating and moving in console windows.

To perform I/O involving one of the ANSI standard streams (`stdin`, `stdout`, `stderr`), simply add the ANSI library to your project, described in Chapter 3, "ANSI." The console package automatically creates a console window and associates these streams to that window by default. You need to add the `console` library to your project only if you explicitly call one of the console functions described in the following catalogue.

Using the console library

To use the console functions, simply add the `console` library to your project and `#include` the `console.h` file in any file that uses console functions.

What Is a Console?

A console is a Macintosh window that behaves like a glass tty or a dumb terminal. ("tty" is an abbreviation for "teletypewriter.") A program can create additional consoles.

How does a program use console functions?

Your program doesn't need to call the console library routines directly. The ANSI library calls the console library automatically in order to perform console-related I/O operations.

If you do want your THINK C program to use the console library routines directly, make sure the program's file `#includes console.h` and make sure the ANSI library is added to your THINK C project.

You can then make calls to the console library routines, and I/O will proceed according to the environment in which the program is running. Environments are explained in the next section.

A generally useful console function is `ccommand()`, which allows you to enter command-line arguments, including directions to redirect `stdin`, `stdout`, and `stderr` to files.

Console routines are not completely portable

Keep in mind that once you explicitly incorporate THINK C console library routines into your programs (as opposed to having ANSI library routines call them automatically), your programs no longer will be portable out of the Macintosh environment.

Environments

You can use the console package in two environments: the generic console environment and the Macintosh environment.

The console package automatically selects between the two environments, partly on the basis of initialization calls that appear in regular Macintosh programs.

Generic Console Environment

In this environment, the console package supplies a default menu bar, and its windows are the only windows.

Programs must be written in plain C (or ported from a Unix or MS-DOS environment). That is, you can not use the Macintosh Toolbox or other Macintosh facilities.

In this environment, consoles can be used for both input and output, as they are in C.

Macintosh Environment

In the Macintosh environment, your program is a Macintosh-style program, as opposed to a plain C program. It has its own event loop, menus, and windows. By using the console package, the program has the additional ability to output to a console window.

The window has whatever menu bar the program gives it. The program can create as many windows as it wishes, and the windows can coexist with other windows from other programs.

In the Macintosh environment, consoles can be used for output only.

Options

To set options for a console, the program sets elements in a global structure called `console_options`. This structure resides in `console.h` and looks like this:

```
struct {
    short    top;
    short    left;
    char     *title;
    short    procID;
    short    txFont;
    short    txSize;
    short    txFace;
    short    nrows;
    short    ncols;
    short    pause_atexit;
};
```

When the program changes the contents of `console_options`, the changes affect the characteristics of the next console to be created.

Remember, once a console is created, its characteristics cannot be changed.

The functions that use the settings in `console_options` are `fopenc()` and `freopenc()`.

Input Modes

Console input can proceed in one of four modes, as follows:

<code>C_ECHO</code>	line-buffered, echo, full editing
<code>C_NOECHO</code>	line-buffered, no echo, partial editing
<code>C_CBREAK</code>	unbuffered, echo, no editing
<code>C_RAW</code>	unbuffered, no echo, no editing

In the line-buffered modes (`C_ECHO` and `C_NOECHO`):

- The console package does not make input available to the program until the program's user types a newline (RETURN or ENTER). (For example, a `readline()` call won't return until the user has entered the whole line.)
- The user may edit the line as they type it using backspace (DELETE) or line-erase (control-U, ESC, CLEAR).
- Control-D signifies keyboard end-of-file. (Useful for emulating a true EOF.)

In the unbuffered modes, the console package makes input available to the program as soon as each character is typed:

- In `C_RAW` mode, the console package does not perform any special interpretation of characters at all. Input requests do not block, that is, if the program calls `getchar()` and the user has not typed any characters, `getchar()` returns EOF immediately.
- In `C_BREAK` mode, the console package makes input available to the program after a single character is typed. In effect, this is as if `C_RAW` mode were echoing.

Video Modes

There are two video modes: normal and inverse.

When your program enables inverse video, the console package cannot use the characters in the upper half of the character set. This is because the console package uses the high bit of each displayed character to denote whether the character should be displayed in normal or inverse video. This also means that the program must make sure that control characters that it wishes to output, such as `'\n'`, do not have their high bits set.

Remember, `cinverse()` does not turn inverse on or off. It only enables inverse video to be turned on or off. Other software must flip the bits in the characters that are to be displayed in order for them to appear in inverse video.

ccleol

LIBRARY

console

SYNTAX

```
#include <console.h>
void ccleol(FILE *fp);
```

DESCRIPTION

Clear to end of line.

`ccleol()` erases characters that follow the cursor and are on the same line that it is on, in the console that `fp` refers to.

ccleos

LIBRARY

console

SYNTAX

```
#include <console.h>
void ccleos(FILE *fp);
```

DESCRIPTION

Clear to end of screen.

`ccleos()` erases the line containing the cursor and all lines that follow it to the end of the screen, in the console that `fp` refers to. `ccleos()` leaves the cursor at the start of the first erased line.

ccommand

LIBRARY

console

SYNTAX

```
#include <console.h>
int ccommand(char ***p);
```

DESCRIPTION

Get command line.

`ccommand()` provides Unix-style I/O redirection and command-line processing.

Call `ccommand()` from `main()` as follows:

```
main(int argc, char **argv)
{
    argc = ccommand(&argv);
    ...
}
```

`ccommand()` displays a dialog that lets you enter command-line arguments and redirect `stdin` and `stdout`.

RETURNS

The number of command-line arguments, `argc`.

cecho2file

LIBRARY

console

SYNTAX

```
#include <console.h>
void cecho2file(char *file, int append, FILE *fp);
```

DESCRIPTION

Echo output to file.

`cecho2file()` echoes to a file named `file` complete lines that any program subsequently writes to the console that `fp` refers to.

If the file exists and `append` is non-zero, `cecho2file()` writes the echoed lines to the end of the file. If the file doesn't exist, `cecho2file()` creates it now and writes the echoed lines to the file.

cgetxy

LIBRARY

console

SYNTAX

```
#include <console.h>
void cgetxy(int *x, int *y, FILE *fp);
```

DESCRIPTION

Report cursor position.

cgetxy () stores the current position of the cursor into *x and *y, where x is for the column and y is for the row, and the upper left corner is at 1, 1. fp refers to the console that the cursor belongs to.

cgotoxy

LIBRARY

console

SYNTAX

```
#include <console.h>
void cgotoxy(int x, int y, FILE *fp)
```

DESCRIPTION

Set cursor position.

`cgotoxy()` positions the cursor to `x` and `y`, where `x` is the column and `y` is the row. The upper left corner is 1, 1. `fp` refers to the console that the cursor belongs to.

Note: `cgotoxy()` does not check its arguments. Be careful not to put the cursor off-screen.

chide

LIBRARY

console

SYNTAX

```
#include <console.h>
void chide(FILE *fp);
```

DESCRIPTION

Hide console.

`chide()` makes invisible the console that `fp` refers to.

cinverse

LIBRARY

console

SYNTAX

```
#include <console.h>
void cinverse(int x, FILE *fp);
```

DESCRIPTION

Enable or disable inverse video mode in the console that `fp` refers to.

`cinverse()` sets the inverse video mode according to the value of `x`:

Zero disables inverse video mode.

Non-zero enables inverse video mode.

`cinverse()` erases the entire window and moves the cursor to the start of its line.

When a console is created, inverse video is disabled.

For more information, read the section "Video Modes" above.

Remember, `cinverse()` does not turn inverse on or off. It only enables inverse video to be turned on or off. Other software must flip the bits in the characters that are to be displayed in order for them to appear in inverse video.

csetmode

LIBRARY

console

SYNTAX

```
#include <console.h>
void csetmode(int mode, FILE *fp);
```

DESCRIPTION

Set input mode.

`csetmode()` sets the input mode of the console. This resetting affects subsequent input, but it does not affect characters that are already buffered.

These are the arguments to `csetmode()`:

<code>mode</code>	Specifies the mode in which the console will operate. mode can have one of four values: <code>C_ECHO</code> line-buffered, echo, full editing <code>C_NOECHO</code> line-buffered, no echo, partial editing <code>C_CBREAK</code> unbuffered, echo, no editing <code>C_RAW</code> unbuffered, no echo, no editing
<code>fp</code>	Pointer to the console.

When a console is created, the input mode is initially `C_ECHO`.

For more information on these modes, read the section "Input Modes" above.

csettabs

LIBRARY

console

SYNTAX

```
#include <console.h>
void csettabs(int n, FILE *fp);
```

DESCRIPTION

Set tab stops.

`csettabs()` sets tab stops every `n` spaces in the console that `fp` refers to.

This resetting of the tab stops affects subsequent output, but it does not change output that is already displayed.

When a console is created, tabs stops are initially set every 8 spaces.

cshow

LIBRARY

console

SYNTAX

```
#include <console.h>
void cshow(FILE *fp);
```

DESCRIPTION

Show and select console.

`cshow()` makes visible the console window that `fp` refers to and brings it to the front of the Macintosh screen.

fopenc

LIBRARY

console

SYNTAX

```
#include <console.h>
FILE *fopenc(void);
```

DESCRIPTION

Open a file.

`fopenc()` creates a new console and opens a new stream on that console.

To set options for the console, set the elements in the global structure, `console_options`. (This structure resides in `console.h` and is described in detail in the section “Options”.)

When the program changes the contents of `console_options`, the changes affect the characteristics of the next console to be created. And remember, once a console is created, its characteristics cannot be changed.

RETURNS

A pointer to a console.

freopenc

LIBRARY

console

SYNTAX

```
#include <console.h>
FILE *freopenc(FILE *fp1, FILE *fp2);
```

DESCRIPTION

Open a console on a specific stream.

First, `freopenc()` closes the stream that `fp2` refers to, if necessary. It then reopens it as a console as follows:

- If `fp1` is `NULL`, `fp2` is a pointer to a newly-created console.
- If `fp1` is not `NULL` and refers to a console, `fp2` will refer to that console, too.

RETURNS

A pointer to a console.

unix 5

This chapter describes THINK C's `unix` library and includes a complete catalogue of the functions it contains.

Topics covered in this chapter:

- What is the Unix library?
- A catalogue of functions

What Is the Unix Library?

The `unix` library implements a few functions common to many Unix implementations of C that were not included in the ANSI standard library. It includes functions to handle files, signals, and other Unix-specific features.

Using the Unix library

To use the `unix` library, add it to your project and `#include` the `unix.h` file in any file that uses its functions.

Do not use the `unix` library in any newly-created programs. Instead, use it to make porting software from Unix systems easier. Its functions have equivalents in the ANSI standard library and the Macintosh system software.

Files

The file-handling functions in the `unix` library are similar to the ones in the ANSI library. The most significant difference between the two sets of functions is how the libraries refer to files. While the ANSI library uses stream pointers, the `unix` library uses small non-negative integers. These numbers are not all related to Macintosh refnums. For compatibility with Unix and MS-DOS, the numbers 0, 1, and 2 refer to `stdin`, `stdout`, and `stderr`, respectively.

Standard Libraries Reference

These Unix and ANSI file-handling functions are roughly equivalent:

Unix library:

open()
close()
read()
write()
fdopen()
creat()
lseek()
tell()

ANSI library:

fopen()
fclose()
fread()
fwrite()
freopen()
fopen()
fseek()
ftell()

close

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int close(int fn);
```

DESCRIPTION

close () closes the file whose file descriptor number is fn.

fn is the number returned by the latest creat () or open () call that opened the file.

RETURNS

0, if successful

-1, if failure.

SEE ALSO

open (), creat (), read (), write () .

creat

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int creat(char *filename, int mode);
```

DESCRIPTION

`creat()` creates and opens a new file. If the file already exists, `creat()` will truncate its size to zero.

`mode` specifies the mode in which `creat()` will open the file. For more information on this argument, see `open()` in this chapter. `creat()` is, in effect, a special case of the `open()` function.

RETURNS

The file descriptor number of the newly created and opened file, if successful

EOF, if failure.

SEE ALSO

`open()`, `close()`, `read()`, `write()`

exec
execl
execle
execlp
execlpe
execv
execve
execvp
execvpe

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int exec(char *path, ...);
int execl(char *path);
int execle(char *path);
int execv(char *path, char argv[]);
int execve(char *path, char *argv[], char *envp[]);
int execvp(char *path, ...);
int execvpe(char *path, ...);
```

DESCRIPTION

These functions launch the file named in path.

These functions ignore all arguments except the path argument (first argument).

RETURNS

Nothing. (A successful exec () call never returns.)

fdopen

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
FILE *fdopen(int fn, char mode);
```

DESCRIPTION

fdopen () returns a pointer to the stream that the file descriptor `fn` refers to. `mode` is ignored.

RETURNS

A pointer to the stream associated with the given file descriptor.

fileno

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int fileno(FILE *stream);
```

DESCRIPTION

`fileno()` returns the file descriptor number of `stream`, where `stream` is a pointer to a file descriptor.

RETURNS

The file number of `stream`, if successful.

EOF, if failure

SEE ALSO

FILE, `open()`, `close()`, `fopen()`, `fclose()`

getegid
geteuid
getgid
getpgrp
getpid
getppid
getuid

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int getegid(void);
int geteuid(void);
int getgid(void);
int getpgrp(void);
int getpid(void);
int getppid(void);
int getuid(void);
```

DESCRIPTION

These functions are provided for Unix compatibility only. In THINK C, they return numbers that are intended to mimic their respective entities but have no real meaning in the Macintosh environment.

Function:	Returned value:
getegid()	Group ID.
geteuid()	User ID.
getgid()	Process's group ID.
getpgrp()	Process group ID of the calling process.
getpid()	Program's process ID number.
getppid()	Calling process's parent's process ID.
getuid()	Program's user ID number.

getlogin

LIBRARY

Unix

SYNTAX

```
#include <unistd.h>
char *getlogin(void);
```

DESCRIPTION

getlogin() returns the user name set in the Chooser desk accessory. (In the Unix environment, getlogin() returns the username under which the program was started.)

RETURNS

The user name set in the Chooser desk accessory.

getw

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int getw(FILE *stream);
```

DESCRIPTION

`getw()` returns the next two bytes of `stream` as an integer.

`getw()` assumes the bytes are in memory order (that is, according to the MC68000 convention, with the MSB in the first byte).

RETURNS

An integer representing the next two bytes of `stream`, if successful

EOF, if failure.

SEE ALSO

`putw()`

isatty

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int isatty(int fn);
```

DESCRIPTION

`isatty()` tests whether or not a stream is associated with a console. This is useful for verifying that a user will see messages that are printed to the stream, and if, on EOF, it makes sense to ask for more input from the stream.

Use `isatty()` this way:

```
isatty(fileno(fn));
```

where `fn` is a pointer to a stream.

RETURNS

Non-zero, if the given stream pointer refers to a console.

Zero, if the given stream pointer refers to anything else.

kill

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int kill(int pid, int sig);
```

DESCRIPTION

In Unix, `kill()` sends the signal `sig` to the process specified by `pid`. However, the Macintosh does not support interprocess communication. So, `kill()` can send the signal to its own application only.

In THINK C, `kill()` raises the signal `sig` if `sig` is not zero and if `pid` is one of the following: 0, -1, the value returned by `getpid()`, or the negative of the value returned by `getpgrp()`. If `pid` is not one of those values, `kill()` returns -1 and sets `errno` to `ESRCH`.

RETURNS

0, if successful.

-1, if failure. It also sets `errno` to `ESRCH`.

SEE ALSO

`signal()` (in the ANSI library), `raise()` (in the ANSI library)

lseek

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
long lseek(int fn, long offset, int whence);
```

DESCRIPTION

`lseek()` sets the file position indicator for the given file. (The file position indicator measures the position in characters from the beginning of the file.) It is much like `fseek()`, except that `lseek()` takes a file descriptor number instead of a pointer to a file. For more information, see the section `fseek()` in the ANSI chapter.

These are the arguments to `fseek()`:

<code>fn</code>	The file descriptor number for the file.
<code>offset</code>	The amount of space in the file through which to seek.
<code>whence</code>	The position from which to start seeking.

These macros are useful values for `whence`:

<code>SEEK_SET</code>	The beginning of the file.
<code>SEEK_CUR</code>	The current value of the file position indicator.
<code>SEEK_END</code>	The end of the file.

RETURNS

The value of the file position indicator, if successful

EOF, if unsuccessful.

SEE ALSO

`fseek()` (in the ANSI library).

open

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int open(char *filename, int mode);
```

DESCRIPTION

`open()` opens a filename for reading or writing, according to mode.

`open()` returns the file descriptor number of the open file.

mode is a flag word that you set by OR-ing together various flags. These flags determine how `open()` will open the file. The flags are defined in `fcntl.h`.

Opening Flags: mode must have exactly one flag from this group:

<code>O_RDONLY</code>	Open this file for reading only.
<code>O_WRONLY</code>	Open this file for writing only.
<code>O_RDWR</code>	Open this file for reading and writing.

Creation Flags: mode may have any number of flags from this group:

<code>O_APPEND</code>	Set the file position indicator to the end of the file prior to each write.
<code>O_CREAT</code>	Create the file if it doesn't already exist.
<code>O_TRUNC</code>	If the file exists, reset its length to 0 (zero). Leave the mode and owner unchanged.
<code>O_EXCL</code>	If the file exists, don't create it again, but do set the file position indicator to the beginning of the existing file.

File-Type Flags: mode may have at most one flag from this group:

<code>O_TEXT</code>	Text file.
<code>O_BINARY</code>	Binary file.

If neither file-type flag is set, `open()` uses the global variable `_fmode`, declared in `fcntl.h`. The initial value of `_fmode` is `O_BINARY`. The user may change this value by assigning it at runtime. The value of `_fmode` does not affect the mode of files opened with `fopen()`.

SEE ALSO

`close()`, `creat()`, `read()`, `write()`

putw

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int putw(int word, FILE *stream);
```

DESCRIPTION

putw() writes word as two bytes — the high byte and then the low byte — to the given output stream.

putw() assumes the bytes are in memory order (that is, according to the MC68000 convention, with the MSB in the first byte).

RETURNS

word, if successful.

EOF, if failure.

SEE ALSO

getw()

read

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int read(int fn, char *buffer, unsigned int count);
```

DESCRIPTION

`read()` tries to read a number of bytes from a file into an array.

These are the arguments to `read()`:

`fn` File descriptor number of the file from which to read.

`buffer` Buffer to read into.

`count` The number of bytes to read.

RETURNS

The number of bytes read, if successful

The number of bytes read before end-of-file, if the end of the file is reached. (Unix systems return 0.)

EOF, if failure.

SEE ALSO

`write()`, `open()`, `close()`

sleep

LIBRARY

Unix

SYNTAX

```
#include <unistd.h>
void sleep(int secs);
```

DESCRIPTION

sleep() waits secs seconds and then continues.

tell

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
long tell(int fn);
```

DESCRIPTION

`tell()` returns the current value of the file position indicator for the file descriptor number `fn`.

RETURNS

The current value of the file position indicator for the given stream `fn`, if successful.

EOF, if failure.

SEE ALSO

`ftell()` (in the ANSI library), `fseek()`, `seek()`

unlink

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int unlink(char *filename);
```

DESCRIPTION

unlink() deletes the file filename.

If the file is open, unlink() will return an error and leave the file unchanged.

RETURNS

0, if successful.

-1, if failure.

write

LIBRARY

Unix

SYNTAX

```
#include <unix.h>
int write(int fn, char *buffer, unsigned nbytes);
```

DESCRIPTION

`write()` copies bytes from a buffer to a file.

These are the arguments to `write()`:

<code>fn</code>	File descriptor number of the file to write to.
<code>buffer</code>	Buffer from which to copy bytes.
<code>nbytes</code>	Number of bytes to copy.

RETURNS

The number of bytes it copied, if successful.

-1, if failure.

SEE ALSO

`read()`, `open()`, `close()`

printf() / scanf() A

This appendix describes the format arguments used with the following functions:

<code>fprintf()</code>	<code>fscanf()</code>
<code>printf()</code>	<code>scanf()</code>
<code>sprintf()</code>	<code>sscanf()</code>
<code>vfprintf()</code>	
<code>vprint()</code>	
<code>vsprintf()</code>	

Topics covered in this chapter:

- Structure of format arguments
- Format arguments for `printf()`.
- Format arguments for `scanf()`.
- Common errors.

Structure of Format Arguments

Each format argument can have a variable number of format specifiers. Each format specifier, in turn, is composed of a variable number of directives, most of which are single characters.

In general, a format specifier must begin with a percent sign (%) and end with a conversion character. Between the % and the conversion character, there may be the following directives, in order:

- Zero or more flags, in any order, that modify the meaning of the rest of the format element.
- A number that specifies the minimum field width into which or out to which the function will put the data.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the number of characters or digits to print.
- An `h`, `l` (letter ell), or `L` that specifies the size of the argument.

To summarize, the format specifiers for `printf()`-related functions have this form (where brackets indicate a choice of optional values):

`% [flags] [field-size] [ .precision] [arg-size] conversion`

And the format specifiers for `scanf()`-related functions have the form (where brackets indicate a choice of optional values):

`% [flags] [field-size] [arg-size] conversion`

The following sections describe the format arguments for `printf()`-related functions and `scanf()`-related functions. Generally, the format arguments behave the same in both scan and print functions. Where differences occur, the text notes them in detail.

Format Arguments for `printf()`

This section describes the format specifiers you can use to construct format arguments for the `printf()`-related functions: `fprintf()`, `printf()`, `sprintf()`, `vfprintf()`, `vprintf()`, and `vsprintf()`.

The format arguments for these functions have the following syntax:

flags *field size* *precision* *arg size* *conversion*
`% [-+<space>#0]    [*<integer>]    [.<integer>]    [hlL]    cdeEfgGinopsuxX%`

The following sections describe each of the format directives in this diagram in detail.

Flag character

A flag character modifies the specification in some way. Any number of flags are allowed in a specification. These are the flags:

Directive	Meaning
- (hyphen)	Left-justifies what's printed within the field.
+ (plus sign)	Prints a positive number with a plus sign. (Otherwise, a sign character is printed only when a number is negative.)
' ' (space)	Prefixes a number with a space, if one of these is true: - The first character of a signed numeric conversion is not a sign. - A signed numeric conversion has no characters. If both the space and + flags appear, the space flag is ignored.

Directive	Meaning
# (pound)	Converts what's to be printed to an alternate form: • For o conversion, increases the precision to force the first digit of the result to be zero. • For x (or X) conversion, prefixes 0x (or 0X) to a nonzero result. • For e, E, f, g, and G conversion, always includes a decimal point in the result, even if no digits follow the decimal point. • For g and G conversions, does not remove trailing zeros from the result. • For s conversions, accepts a Pascal-style string (an object of type <code>Str255</code>). This is a THINK C extension.
0 (zero)	Pads the field with leading zeros (inserting them after any indication of sign or base) for d, i, o, u, x, X, E, f, g, and G conversions. For d, i, o, u, x, and X conversions, if there's precision specification, the 0 flag is ignored. If both the 0 and - flags appear, the 0 flag is ignored.

Field size

The field size directive specifies the width of the field into which the function will print the converted data argument. The field size directive can be one of these:

Directive	Function
* (asterisk)	Uses the next directive in the format specifier, which must be an integer, as the size for the field size into which it will print the data.
an integer	Uses this integer as the size for the field into which it will print the data.

If the data the function prints has fewer characters than the field size, the function will pad the output with spaces on the left or right, depending on what flags are present.

Precision directive

The precision directive specifies one of these, depending on the argument that follows:

Arg.	The precision directive specifies...
string	The maximum number of characters to print from the string.
floating-point number	The number of digits after the decimal point of the floating-point value to print.
integer	The minimum number of digits to print.

The precision directive is a period (.) followed optionally by one of the following:

Directive	Meaning
a decimal integer	Use this integer as the precision.
* (asterisk)	Use the next argument as the precision specifier. That argument must be an integer.

If neither an integer nor an asterisk appear, the function assumes a precision of 0.

Note: When an asterisk appears in the field size and precision directives of a format specifier, make sure the arguments to which they refer appear immediately after the format argument in this order: field size, precision, and the argument to print.

Argument size

The argument size specifies the size of the argument. It can be one of the following:

Directive	Meaning
h	This d, i, o, u, x, or X conversion specifier applies to an argument of type <code>short int</code> or unsigned <code>short int</code> . Converts the argument before printing, if necessary.
l (letter ell)	This d, i, o, u, x, or X conversion specifier applies to an argument of type <code>long int</code> or an unsigned <code>long int</code> . Converts the argument before printing, if necessary.
L	This e, E, f, g, or G conversion specifier applies to an argument of type <code>long double</code> . Converts the argument before printing if necessary.

h, l, and L are not valid with any other conversion specifiers.

Conversion characters

The conversion character specifies how the argument should be printed. These are the conversion characters:

Character	Argument Type	Output
c	int	A single character
d	int	A signed decimal integer. [-] <i>dddd</i>
e, E	double	A signed decimal floating point number. [-] <i>m.ddde±xx</i> (for e) [-] <i>m.dddE±xx</i> (for E) - Prints one digit before the decimal point. - Prints the number of digits after the decimal point specified by the precision directive. The default is 6. - If the precision directive is 0 and there is no # flag in this format specifier, this does not print a decimal point. - Rounds the value to the appropriate number of digits.
f	double	A signed decimal fixed-point number. [-] <i>mmm.ddd</i> - Prints the number of digits after the decimal point specified by the precision directive. The default is 6. - If the precision directive is 0 and there is no # flag in this format specifier, this does not print a decimal point. - If this prints a decimal point, it prints at least one digit before the decimal point. - Rounds the value to the appropriate number of digits.
g	double	Uses the f or e format, whichever is smaller
G	double	Uses the f or E format, whichever is smaller.

Character	Argument Type	Output
i	int	A signed decimal integer. <i>[-] dddd</i> • Pads the result with leading zeros if there are fewer digits than the precision directive allows. • The default precision is 1. • If the value is zero and the precision is zero, this prints nothing.
n	int *	Nothing printed. This puts into an integer the number of characters the function has printed so far during this call.
o	unsigned int	An unsigned octal integer. <i>ddd</i> • Pads the result with leading zeros if there are fewer digits than the precision directive allows. • The default precision is 1. • If the value is zero and the precision is zero, this prints nothing.
p	void *	An eight-digit hexadecimal value.
s	char * Str255	A string. Without the # flag, this specifier prints a C-style string (an object of type <code>char *</code>). It prints the string until one of the following happens: • It encounters a NULL character, which it will not print. • It prints the maximum number of characters allowed by the precision directive. With the # flag, this specifier prints a Pascal-style string (an object of type <code>Str255</code>).
u	unsigned int	An unsigned decimal integer. • Pads the result with leading zeros if there are fewer digits than the precision directive allows.

Character	Argument Type	Output
x, X	unsigned int	An unsigned hexadecimal integer. • If the directive is x, this uses a, b, c, d, e, and f as the extra digits. • If the directive is X, this uses A, B, C, D, E, and F as the extra digits. • Pads the result with leading zeros if there are fewer digits than the precision directive allows.
%	No argument used	A percent sign (%)

Format Arguments for scanf()

This section describes the format specifiers you can use to construct format arguments for the `scanf()`-related functions: `fscanf()`, `scanf()`, and `sscanf()`.

The format arguments for these functions have this syntax:

```

    flag    field size    arg size    conversion
% [*]    [<integer>] [hLl]    cdefginopsux% [...]

```

The following sections describe each of the format specifiers in this diagram in detail.

Flag character

The flag character controls whether the input field that matches the specifier is ignored.

Directive	Meaning
* (asterisk)	Suppresses the assignment of the input field that matches this specifier.

Field size

The field size directive specifies the width of the input field.

Directive	Meaning
a decimal integer	Specifies the width of the field from which this conversion specifier will read the converted data argument(s).

Argument size

The argument size directive specifies the size of the argument. These are the argument size directives:

Directive	Meaning
h	Specifies that this conversion specifier applies to an argument of the following type: • For d, i, or n specifiers, <code>short int *</code>. • For o, u, or x specifiers, <code>unsigned short int *</code>.
l (letter ell)	Specifies that this conversion specifier applies to an argument of the following type: • For d, i, or n specifiers, <code>long int *</code>. • For o, u, or x specifiers, <code>unsigned long int *</code>. • For e, f, or g specifiers, <code>double *</code>. This can also specify that the n conversion specifier applies to an object of type <code>long int *</code> .
L	Specifies the e , f , or g conversion specifier applies to an argument that is pointer to a <code>long double</code> .

h, **l**, and **L** are not valid with any other conversion specifiers.

Conversion characters

The conversion character specifies how to interpret the input field. These are the conversion characters:

Character	Argument Type	Input
c	<code>char *</code>	A single character • Accepts all kinds of characters: white space, control characters, etc. • The pointer must point to an object large enough to hold all the characters specified. • Does not add a terminating <code>NULL</code>.
d	<code>int *</code>	A decimal integer
e , f , g	<code>float *</code>	A floating-point number with optional sign, decimal point, and exponent integer.

Character	Argument Type	Input
i	int *	An integer Figures the base of the number according to its appearance: • Leading 0 means it's octal (base 8). • Leading 0x or 0X means it's hexadecimal (base 16). • Otherwise, it's decimal (base 10).
n	int *	No argument read. Puts into its argument the number of characters the function has read so far during this call.
o	unsigned int *	An octal integer.
p	void *	A hexadecimal number.
s	char *	A character string. • Reads only non-white-space characters. • The pointer must point to a memory area that is large enough to hold all the characters. • Adds a terminating NULL.
u	unsigned int *	A decimal integer.
x	unsigned int *	A hexadecimal integer.
%	No argument filled	A percent sign (%).
[...]	char *	Read section "Scan sets" below.

Scan sets

Scan sets allow you to determine exactly which characters are read into a string argument. In a scan set of the form % [...] you can list just the characters you want to read. In a set of the form % [X-X] you can give a range of characters you want to read. The range is determined using the ASCII ordering sequence. And in a set of the form % [^...] you can specify the characters you do *not* want to read.

Here are several example scan sets:

Scan set	Read the longest string of characters in which...
<code>%[xyz]</code>	all are x, y, or z.
<code>%[^xyz]</code>	none are x, y, or z.
<code>%[]xyz]</code>	all are x, y, z, or the right bracket.
<code>%[^]xyz]</code>	none are x, y, z, or the right bracket.
<code>%[a-z]</code>	all are lowercase alphabetic characters (from a to z).
<code>%[^a-z]</code>	none are lowercase alphabetic characters (from a to z).
<code>%[-xyz]</code>	all are x, y, z, or the hyphen.
<code>%[xyz-]</code>	all are x, y, z, or the hyphen.
<code>%[^-xyz]</code>	none are x, y, z, or the hyphen.

Common Errors

- Make sure there is one variable argument of the correct size for each format specifier in format.
- Make sure all arguments are of the right type.
- A common misconception about `printf()` is that `printf()` “knows” about its arguments. If you pass a long expression to `printf()`, you must specify in the format string that you want a long expression to be printed. Example:

```
int anInt;
long aLong;
printf("long is %ld, int is %d\n", aLong, anInt);
```

In `scanf`, the same goes for floats and doubles:

```
float aFloat;
double aDouble;
printf("Input a double and a float:");
scanf("%lf %f", &aDouble, &aFloat);
```

Index

Entries in **bold face** refer to menu commands. Entries in typewriter face refer to functions, methods, variables, keywords, or files.

printf directive 205
%#s 208
%% 209, 211
%[...] 211
%c 207, 210
%d 207, 210
%e 207, 210
%f 207, 210
%g 207, 210
%i 208, 211
%n 208, 211
%o 208, 211
%p 208, 211
%s 208, 211
%u 208, 211
%x 209, 211
*
 printf directive 205, 206
 scanf directive 209
+,printf directive 204
-,printf directive 204
_atexit 25
_exit 26
_exiting 27
_IOFBF 17
_IOLBF 17
_IONBF 17
abort 28
abs 29
acos 30
ANSI library 23
ANSI-881 library 24
ANSI-A4 library 24
ANSI-small library 24
asctime 31
asin 32
assert 33
assert.h 4
atan 34
atan2 35
atexit 36
atof 37
atoi 38
atol 39
bsearch 40
BUFSIZ 16
C_CBREAK 167
C_ECHO 167
C_NOECHO 167
C_RAW 167
calloc 41
ccleol 169
ccleos 170
ccommand 171
cecho2file 172
ceil 42
cgetxy 173
cgotoxy 174
CHAR_BIT 9
CHAR_MAX 9
CHAR_MIN 9
chide 175
cinverse 176
clearerr 43
clock 44
clock_t 21
close 185
console 165
 input modes 167
 inverse mode 168
 normal mode 168

Standard Libraries Reference

- options 167
- video modes 168
- console environment 166
 - Macintosh 166
 - generic 166
- cos 45
- cosh 46
- creat 186
- csetmode 177
- csettabs 178
- cshow 179
- ctime 47
- ctype.h 5
- DBL_DIG 7
- DBL_EPSILON 8
- DBL_MANT_DIG 7
- DBL_MAX 8
- DBL_MAX_10_EXP 8
- DBL_MAX_EXP 7
- DBL_MIN 8
- DBL_MIN_10_EXP 7
- DBL_MIN_EXP 7
- difftime 48
- div 49
- div_t 19
- EDOM 6
- ERANGE 6
- errno 6
- errno.h 6
- exec 187
- execl 187
- execle 187
- execlp 187
- execlpe 187
- execv 187
- execve 187
- execvp 187
- execvpe 187
- exit 50
- EXIT_FAILURE 19
- EXIT_SUCCESS 19
- exp 51
- fabs 52
- fclose 53

- fdopen 188
- feof 54
- ferror 55
- fflush 56
- fgetc 57
- fgetpos 58
- fgets 59
- FILE 16
- FILENAME_MAX 16
- fileno 189
- float.h 7
- floor 60
- FLT_DIG 7
- FLT_EPSILON 8
- FLT_MANT_DIG 7
- FLT_MAX 8
- FLT_MAX_10_EXP 8
- FLT_MAX_EXP 7
- FLT_MIN 8
- FLT_MIN_10_EXP 7
- FLT_MIN_EXP 7
- FLT_RADIX 7
- FLT_ROUNDS 7
- fmod 61
- fopen 62
- FOPEN_MAX 16
- fopenc 180
- format specifiers 203-212
- fpos_t 16
- fprintf 64, (see also printf)
- fputc 65
- fputs 66
- fread 67
- free 68
- freopen 69
- freopenc 181
- frexp 70
- fscanf 71, (see also scanf)
- fseek 72
- fsetpos 73
- ftell 74
- fwrite 75
- getc 76

```

getchar 77
getegid 190
getenv 78
geteuid 190
getgid 190
getlogin 191
getpgrp 190
getpid 190
getppid 190
getuid 190
getw 192
gmtime 79
h
    printf directive 206
    scanf directive 210
HUGE_VAL 11
Infinity 11
INT_MAX 9
INT_MIN 9
isalnum 80
isalpha 80
isatty 193
iscntrl 80
isdigit 80
isgraph 80
islower 80
isprint 80
ispunct 80
isspace 80
isupper 80
isxdigit 80
jmp_buf 12
kill 194
l
    printf directive 206
    scanf directive 210
L_tmpnam 16
labs 82
LC_ALL 10
LC_COLLATE 10
LC_MONETARY 10
LC_NUMERIC 10
LC_TIME 10
LC_TYPE 10
lconv 10
LDBL_DIG 7
LDBL_EPSILON 8
LDBL_MANT_DIG 7
LDBL_MAX 8
LDBL_MAX_10_EXP 8
LDBL_MAX_EXP 7
LDBL_MIN 8
LDBL_MIN_10_EXP 7
LDBL_MIN_EXP 7
ldexp 83
ldiv 84
ldiv_t 19
limits.h 9
locale.h 10
localeconv 85
localtime 86
log 87
log10 88
LONG_MAX 9
LONG_MIN 9
longjmp 89
lseek 195
malloc 90
math.h 11
MB_CUR_MAX 19
MB_LEN_MAX 9
mblen 91
mbstowcs 92
mbtowc 93
memchr 94
memcmp 95
memcpy 96
memmove 97
memset 98
mktime 99
modf 100
NULL 15, 16, 20, 21
O_APPEND 196
O_BINARY 196
O_CREAT 196
O_EXCL 196
O_RDONLY 196

```

Standard Libraries Reference

O_RDWR 196
O_TEXT 196
O_TRUNC 196
O_WRONLY 196
offsetof 15
open 196
perror 101
pow 102
printf 103
 argument size 206
 conversion characters 207-209
 field size 205
 flag characters 204
 format arguments 204-209
 format structure 203
 format syntax 204
 precision 206
ptrdiff_t 15
putc 104
putchar 105
puts 106
putw 197
qsort 107
raise 108
rand 109
RAND_MAX 19
read 198
realloc 110
remove 111
rename 112
rewind 113
scan sets 211
scanf 114, 209
 argument size 210
 conversion characters 210-211
 field size 209
 flag character 209
 format arguments 212
 format arguments 209
 format structure 204
 format syntax 209
 scan sets 211
SCHAR_MAX 9
SCHAR_MIN 9
SEEK_CUR 17
SEEK_END 17
SEEK_SET 17
setbuf 115
setjmp 116
setjmp.h 12
setlocale 117
setvbuf 118
SHRT_MAX 9
SHRT_MIN 9
sig_atomic_t 13
SIG_DFL 13
SIG_ERR 13
SIG_IGN 13
SIGABRT 13
SIGFPE 13
SIGILL 13
SIGINT 13
signal 119
signal.h 13
SIGSEGV 13
SIGTERM 13
size_t 15, 16, 19, 20
sizeof 15
sleep 199
sprintf 121, (see also printf)
srand 122
sscanf 123, (see also scanf)
stdarg.h 14
stddef.h 15
stderr 17
stdin 17
stdio.h 16
stdlib.h 19
stdout 17
strcat 124
strchr 125
strcmp 126
strcoll 127
strcpy 128
strcspn 129
strerror 130

strftime 131
string.h 20
strlen 134
strncat 135
strncmp 136
strncpy 137
strpbrk 138
strrchr 139
strspn 140
strstr 141
strtod 142
strtok 143
strtol 144
strtoul 146
strxfrm 147
system 148
tan 149
tanh 150
tell 200
time 151
time.h 21
time_t 21
tm 21
TMP_MAX 16
tmpfile 152
tmpnam 153
tolower 154
toupper 155
UCHAR_MAX 9
UINT_MAX 9
ULONG_MAX 9
ungetc 156
unlink 201
USHRT_MAX 9
va_arg 157
va_end 157
va_list 14
va_start 157
vfprintf 159, (see also
printf)
vprintf 160, (see also printf)
vsprintf 161, (see also
printf)
wchar_t 15, 19
wcstombs 163
wctomb 162
write 202

License Agreement

License Agreement

This manual and the software described in it were developed and are copyrighted by Symantec Corp. (Symantec) and are licensed to you on a non-exclusive, non-transferable basis. Neither the manual nor the software may be copied in whole or in part except as follows:

- 1) You may make backup copies of the software for your use provided that they bear Symantec's copyright notice.
- 2) You have the right to include object code derived from the libraries in programs that you develop using the software and you also have the right to use, distribute and license such programs to third parties without payment of any further license fees, so long as a copyright notice sufficient to protect *your* copyright in the software in the United States or any other country is included in the graphic display of your software and on the labels affixed to the media on which your software is distributed.

You may not in any event distribute any of the source files provided or licensed as part of the software. You may use the software at any number of locations so long as there is no possibility of it being used at more than one location at a time.

Symantec's Plain Language License Statement

Symantec is concerned with how you copyright your software only in the case where you use object code of libraries which Symantec provides in source form (MacTraps does not fall into this category). These libraries may be included in your program so long as a copyright notice that will protect *your* copyright in the software is in the "About box" of your software and on the disk labels, as specified in the license agreement. You are not required to include a specific Symantec copyright notice except if your copyright does not satisfy the above requirement. This is only an explanation of the License Agreement. All terms and conditions of the License Agreement apply.

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed or in the manuals distributed with the software, Symantec will replace the media or manuals at no cost to you provided that you return the defective materials along with a copy of your receipt to Symantec or to an authorized Symantec dealer during the 60-day period following your receipt of the software.

Limited Warranty on the Product

Symantec warrants that the software will perform substantially as described in the User's Manual. If within 60 days of receiving the software, you give written notification to Symantec

Standard Libraries Reference

of a significant, reproducible error in the software which prevents operation, and provide a written description of the possible problem along with a machine readable example, if appropriate, Symantec will either provide you with corrective or workaround instructions, a corrected copy of the software, a correction to the User's Guide and Reference Manual, or Symantec will refund your purchase price upon return of all copies of the software and documentation together with a copy of your receipt. This warranty extends only to you and shall be void if the software has been tampered with, modified, or improperly used, or if the software is used on hardware other than the Apple Macintosh™ Computer.

EXCEPT FOR THE LIMITED WARRANTY DESCRIBED ABOVE, THERE ARE NO WARRANTIES TO YOU OR ANY OTHER PERSON OR ENTITY FOR THE PRODUCT EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. ALL SUCH WARRANTIES ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. Some states do not allow the exclusion of implied warranties or limitations on how long they last, and you also may have other rights that vary from state to state. IN NO EVENT SHALL SYMANTEC BE RESPONSIBLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR SIMILAR DAMAGES OR LOST DATA OR PROFITS TO YOU OR ANY OTHER PERSON OR ENTITY REGARDLESS OF THE LEGAL THEORY, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. The warranty and remedies set forth are exclusive and in lieu of all others, oral or written, express or implied.

SYMANTEC MAKES NO WARRANTY OF THE PERFORMANCE OF THE LIBRARIES WHEN USED IN YOUR SOFTWARE. YOU AGREE TO INDEMNIFY SYMANTEC FROM ALL CLAIMS BY THIRD PARTIES ARISING IN CONNECTION WITH THE USE OF YOUR SOFTWARE.

General Terms This license states the entire agreement between the parties and supersedes all other communications between the parties relating to this License, which shall be governed and construed in accordance with the laws of the State of California. You agree to bring any proceeding to enforce or construe this License or involving the performance of the software only in a federal or state court residing in the State of California. The prevailing party in any such proceedings shall be entitled to recover its attorneys' fee and litigation expenses in addition to other appropriate relief. If any provision of this License by Symantec shall be held to be unenforceable such holding shall not affect the enforceability of any other provision hereof. Waiver of any breach of this License by Symantec shall not be considered a waiver of any other or subsequent breach. The licensed software is a unique and valuable asset of Symantec and Symantec has the right to seek whatever equitable and legal redress which may be available to it for your breach of the provisions of the License.



SYMANTEC™

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
(408) 253-9600